ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

DOKTORI (PHD) ÉRTEKEZÉS

PINTÉR ÁDÁM

Forráskódok nyelvfüggetlen analízise GPU-n, gépi tanuláson alapuló technikával

Témavezetők:
Prof. Dr. Szénási Sándor
Prof. Dr. Ludányi-Laufer Edit

ALKALMAZOTT INFORMATIKAI ES ALKALMAZOTT MATEMATIKAI DOKTORI ISKOLA

Budapest, 2026. május 26.

Komplex vizsgabizottság:

Elnök:
Prof. Dr. Horváth László

Tagok:
Prof. Dr. habil. Takács Márta
Prof. Dr. habil. Fullér Róbert

Nyilvános védés teljes bizottsága:

Opponensek:
Dr. Végh László
Dr. Hajnal Ákos László

Elnök:
Prof. Dr. habil. Tar József Kázmér

Tagok:
Prof. Dr. habil. Takács Márta
Prof. Dr. habil. Fullér Róbert
Dr. Odry Ákos

Titkár:
Dr. Simon-Nagy Gábor

Nyilvános védés időpontja:

2026. 07. 06.

Tartalomjegyzék

Nyilatkozat

Alulírott **Pintér Ádám** kijelentem, hogy ez a doktori értekezés a saját munkámat mutatja be, és a saját eredeti kutatásom eredménye. Csak a hivatkozásokban felsorolt forrásokat használtam fel, minden más munkából származó rész, a szó szerinti, vagy az eredeti jelentést megtartó átiratok egyértelműen jelölve, és forrás hivatkozva lettek.

.....

Pintér Ádám

Köszönetnyilvánítás

Ezúton mondok köszönetet témavezetőimnek, Prof. Dr. Szénási Sándornak és Prof. Dr. Ludányi-Laufer Editnek eredményeim elérésében, értekezésem elkészítésében nyújtott magas színvonalú, folyamatos és áldozatos segítségükért.

Köszönöm az Alkalmazott Informatikai és Alkalmazott Matematikai Doktori Iskola tagjainak, különösen Prof. Dr. Galántai Aurélnak és Prof. Dr. Tar Józsefnek a hasznos szakmai tanácsokat és a kollegiális segítséget.

Az Óbudai Egyetemen dolgozó közvetlen kollégáimnak - különösen Prof. Dr. Vámossy Zoltánnak - köszönöm a támogatást, a kitartó ösztönzést értekezésem elkészítése során.

Köszönöm családomnak a megértést, bíztatást és az áldozatvállalást, amivel az értekezésem elkészítését lehetővé tették.

1 Bevezetés

1.1 A téma jelentősége és aktualitása

A technológia fejlődésének köszönhetően napjainkban már szinte mindenhol találkozhatunk forráskódokkal, melyek elemzése a hasonlóságuk tekintetében több területen is nélkülözhetetlen. Egyik ilyen terület a forráskódok klónozásának („code clone") vizsgálatával foglalkozik, amely elsősorban a nagyobb rendszerek fejlesztése során segít elkerülni az azonos funkcionalitással rendelkező kódok duplikálását. Másik területe a forráskódok elemzésének, mikor plagizálás szempontjából kerül vizsgálatra a forrásállomány.

1.1.1 A forráskódok klónozásának terminológiája

A forráskódok klónozásfelderítésének alapja a szövegalapú összehasonlítás. A klónozáshoz kapcsolódó definíciót 2007-ben Bellon és társai [1] írták le, mely meghatározta a kódklónok típusát a hasonlóság mértékére és típusára alapozva. A definíciót azóta számos irodalomban [2,3,4,5] feldolgozták és kiegészítették, létrehozva így a következő terminológiát:

- Kód töredék (*Code Fragment – CF*): kódsor vagy kódsorok, kommenttel vagy anélkül. Legalsó logikai szintje a terminológiának, amelynek másolásáról beszélhetünk, például: utasítás blokkok, függvények. A CF-et egy fájlnevet, kezdősort és végsort tartalmazó hármassal szokás azonosítani az eredeti forráskódban.
- Kód klón (*Code Clone – CC*): egy CF_1 kód klónja egy másik CF_2 kódnak, ha $f(CF_1) = f(CF_2)$, ahol *f* a hasonlóságot leíró függvény.
- Klón pár (*Clone Pair – CP*): két *f* függvény által hasonlónak ítélt CF.
- Klón osztály, Klón csoport (*Clone Group – CG*): számos *f* függvény által hasonlónak ítélt CF-ek csoportja.
- Klón típus (*Clone Type – CT*): a kód klónozásának típusa, melyből az első 3 típus a szöveges, az utolsó pedig a funkcionális hasonlóságot definiálja:
    - *1. típus*: teljesen azonos CF-ek bármilyen módosítás nélkül

(kivéve a szóközöket és a kommenteket).

- *2. típus*: szintaktikailag azonos CF-ek, kivételt képeznek ez alól a változó nevek, adattípusok stb. módosításai.
- *3. típus*: másolt CF-ek további módosításokkal, beleértve a hozzáadott vagy eltávolított utasításokat vagy a további módosításait a kód elrendezésének, azonosítóinak, kommentjeinek.
- *4. típus (funkcionális hasonlóság)*: Kettő vagy több CF, amelyek ugyanazt a végrehajtást eredményezik, de különböző szintaktikai formában valósulnak meg.

1.1.2 A forráskódok klónozásának detektálása

Prajila 2013-as [6] cikkében a forráskódklónozás detektálásának 4 fő típusát definiálta, melyről átfogó képet az **1.1.1.** táblázat ad. A kódklónozás detektálásának 4 fő és 2 hibrid típusa a következő:

- *Szöveges megközelítés*. Az összehasonlítás olyan szövegalapú megközelítés, amely kevés vagy semmilyen transzformációt nem alkalmaz a forráskódon, azt nyers formában használják a klónfelderítés folyamatában.
- *Lexikális megközelítés*. A forráskódokat átalakítják tokenek sorozatává, amelynek egyes részsorozatait vizsgálva eredményként detektálható a duplikált kódsorozat.
- *Szintaktikai megközelítés*. Ez a megközelítés először egy elemzést hajt végre a forráskódon, amelynek következtében a forráskódot egy absztrakt szintaxis fává (AST) fogják leképezni. Ezek a fák már fa egyezést vizsgáló algoritmusokkal vagy strukturális metrikákkal vizsgálhatóak kódklón detektálás céljából.
- *Szemantikai megközelítés*. A forráskódokat statikus programelemzésnek vetik alá, mely így pontosabb információt fog szolgáltatni, mint az egyszerű szintaktikai megközelítések. Gyakori ábrázolási módja a forráskódnak a szemantikai vizsgálat során a program függőségi gráf (PDG), amelyben a csomópontok a kifejezéseket, az élek pedig az adatfüggőséget reprezentálják.

- *Metrika alapú megközelítés*. A forráskódból először egy AST-t vagy CFG-t (control flow graph) készítenek, majd ennek a struktúrának az elrendezéséből, a kifejezésekből és a funkciók vezérlési folyamataiból számítanak különféle metrikákat. A klón úgy definiált ezesetben, mint egy teljes funkcionális pár, amely hasonló mérési értékekkel rendelkezik. Webes dokumentumok és weboldalak klónozásának detektálásánál használják leginkább.
- *Hibrid megközelítés*. A különlegesebb nyelvek esetében, mint amilyen a Lisp is, több módszert szoktak alkalmazni a pontosabb eredmény elérése érdekében. Jellemzően szintaktikai és szemantikai megközelítést használnak valamilyen speciális összehasonlító funkcióval.

1.1.1. táblázat. *Forráskód klónok és technikák osztályozása.*

	Szöveg alapú	**Token alapú**	**AST alapú**	**PDG alapú**
Kategória	Szöveges megközelítés	Lexikális megközelítés	Szintaktikai megközelítés	Szemantikai megközelítés
Klón típusa	1. típus	1., 2. típus	1., 2., 3. típus	1., 2., 3., 4. típus
Komplexitás	$O(n)$	$O(n)$	$O(n)$	$O(n^3)$
n jelentése	kódsorok száma	tokenek száma	AST csomópontok száma	PDG csomópontok száma

1.2 Célkitűzések és kutatási kérdések

A kutatásom célja egy olyan új módszertani megközelítés kidolgozása és bemutatása, amely hozzájárul a forráskódok közötti hasonlóság pontosabb meghatározásához. A szoftverfejlesztésben és kódelemzésben egyre fontosabb szerepet kapnak az automatikus összehasonlító algoritmusok, amelyek segítségével lehetőség nyílik plágiumvizsgálatra, vagy különböző implementációk közötti tartalmi egyezések azonosítására, a programkód tényleges futtatása nélkül.

A forráskódok összehasonlítása számos kontextusban kulcsfontosságú: a plágiumdetektálásnál például nem elegendő a szintaktikai azonosságokat felismerni, hanem a strukturális, logikai egyezőségeket is detektálni kell. Ugyanez igaz az oktatási környezetekre, ahol a programozási feladatokra beadott megoldások összevetésénél fontos figyelembe venni, hogy a hallgatók különböző sorrendben, különféle megfogalmazásban, de gyakran azonos algoritmikus tartalommal oldják meg a feladatokat.

Egy ilyen típusú módszer alkalmazható lehet továbbá kódbázisok karbantartása és refaktorálása során is, például duplikált kódrészek automatikus azonosítására, ahol a pontos karakterazonosság helyett a funkcionális hasonlóság felismerése a lényeg. Nyelvfüggetlensége miatt beépíthető különböző nyelvű szoftverfejlesztő eszközökbe és analitikai rendszerekbe, amelyeknek célja a kódminőség javítása vagy a fejlesztési idő csökkentése. Emellett, a módszer potenciálisan hasznos lehet nyílt forráskódú projektek összehasonlító vizsgálatában is, például különböző verziók vagy forkok összevetésekor, ahol az eltérő szerkezetű, de részben közös kódelemek feltárása komoly jelentőséggel bír. Az is fontos szempont, hogy mivel a megközelítés nem támaszkodik nyelvspecifikus előfeldolgozásra, jól alkalmazható lehet nagy volumenű, heterogén adathalmazokon is, például különböző programozási versenyek, közösségi kódtárak vagy kódgeneráló rendszerek kimeneteinek elemzése során.

A kutatás célkitűzése egy olyan megoldás kialakítása, amely a meglévő, főként természetes nyelvi szövegekre optimalizált metrikáknál alkalmasabban kezeli a programkódokra jellemző szerkezeti sajátosságokat, különösen a független utasítások sorrendjének megváltoztathatóságát, valamint a nyelvspecifikus elemek előfeldolgozás nélküli kezelését.

A kutatás legfőbb kérdése, hogy lehetséges-e olyan módszer kidolgozása, mely képes figyelembe venni az összehasonlítás során a forráskódok sajátosságait, ezáltal pontosabb hasonlósági értéket adva a meglévő metrikáknál, illetve lehetséges-e a módszer adatpárhuzamos gyorsítása a nagyobb adathalmazok feldolgozása érdekében.

2 Nem gépi tanuláson alapuló módszerek

2.1 Forráskódok összehasonlítása LCS alapon

2.1.1 Bevezetés

A forráskódok a természetes nyelvektől eltérően lazább szórenddel rendelkeznek. Habár az egyes struktúrák kötöttek, az azok közötti sorrend tetszőlegesen változtatható (amennyiben független utasításokként állnak elő). Ez a kötetlenség az, amely a hagyományos szövegalapú módszerek használatát nehézkessé teszi és igényli egy olyan megközelítés kidolgozását, amely olyan esetben is alkalmazható a hasonlóság megállapítására, mikor csak az utasítások sorrendje változik.

Vegyük a **2.1.1.** ábrán látható forráskód részleteket, melyek C# programozási nyelven egy tömb elemeit összegzik, illetve összeszorozzák és ennek eredményét megjelenítik a képernyőn. A metódusok funkcionalitása azonos, az első kettő esetében minimális átnevezések és átrendezések történtek, míg a harmadik változatban az összeadás és a szorzás Linq technikával került megvalósításra. Funkcionálisan azonosak a metódusok, azonban a felépítésük merőben eltérő. Ahhoz, hogy a felépítésük alapján eldönthessük, hogy azonosak-e, célszerű megvizsgálni a gyakran használt szöveg alapú hasonlósági metrikákat és az azok által adott hasonlósági értékeket.

2.1.2 A szakirodalomban elterjedt módszerek

A természetes nyelveknél használt gyakori szöveg alapú hasonlósági metrikák három kategóriába sorolhatók, attól függően, hogy milyen alapon számítják a hasonlósági értéket: [7,8]

- *Szerkesztési távolság alapú módszerek.* Ezek a metrikák két karakterlánc közötti hasonlóságot számszerűsítik azáltal, hogy megmérik az egyik karakterlánc másikká alakításához szükséges minimális szerkesztési műveletek számát. A szerkesztési műveletek jellemzően a karakterek beszúrását, törlését és helyettesítését foglalják magukban. Gyakori szerkesztés alapú metrikák: Normalizált-Levenshtein (hasonlóság), Jaro-Winkler (hasonlóság).

```
static void SumAvg(int n, int[] ar)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += ar[i];
    Console.WriteLine(sum);

    int prod = 1;
    for (int i = 0; i < n; i++)
        prod *= ar[i];
    Console.WriteLine(prod);
}
```

*(**A**) Az „eredeti” metódus.*

```
static void SumAvg(int[] ar)
{
    int s = 0;
    foreach (var element in ar)
        s += element;

    int p = 1;
    foreach (var element in ar)
        p *= element;

    Console.WriteLine($"{s},{p}");
}
```

*(**B**) Az első átalakított metódus, melyet úgy kapunk, hogy a sum és a prod változót az A metódusban s és p változóra cseréljük, illetve foreach ciklust használunk a for ciklus helyett. Végezetül az eredmény megjelenítésére használt két Console.WriteLine metódus is összevonásra került.*

```
static void SumAvg(int[] ar)
{
    Console.WriteLine(ar.Sum());
    Console.WriteLine(ar.Aggregate(1, (x, y) => x * y));
}
```

*(**C**) Ebben a változatban a tömb elemeinek összegének és szorzatának meghatározása Linq technika segítségével történik az eredeti ciklusok helyett.*

***2.1.1. ábra.** Egy tömb elemeinek összeadásának és szorzásának különböző megvalósításai. Mindhárom metódus eredménye megegyezik, vagyis ugyanazt a problémát oldják meg, azonban a számítás menete, a felhasznált technikák különböznek.*

- *Token alapó módszerek.* A token alapú metrikák magukba foglalják a karakterláncok kisebb egységekre (tokenekre) bontását, majd ezen tokenek összehasonlítását a karakterláncok közötti hasonlóság meghatározása érdekében. A tokenek lehetnek egyedi karakterek, szavak vagy n-grammok (n darab karakterből vagy szóból álló sorozatok). Ezeket az algoritmusokat széles körben használják természetes nyelvű feldolgozási (NLP) feladatokban, például helyesírás ellenőrzésnél és javításnál, illetve plágium detektálás során. Hagyományos token alapú metrikák: Cosine Similarity (koszinusz hasonlóság), Jaccard-index (hasonlóság), Sorensen-Dice együttható (hasonlóság).
- *Szekvencia alapú módszerek.* A szekvencia alapú metrikák a karakterláncok karaktereinek vagy tokenjeinek sorozatának

számszerűsítésével határozzák meg a hasonlóságot. Legelterjedtebb szekvencia alapó módszer: Longest Common Subsequence (távolság).

Szöveg alapú metrikák között szokás megkülönböztetni a hasonlóságot és a távolságot. A hasonlóság azt méri, hogy két objektum (pl. szöveg, dokumentum, vektorra alakított mondat) mennyire hasonlít egymásra. Minél nagyobb a hasonlósági érték, annál közelebb állnak egymáshoz. A távolság ezzel szemben azt mutatja meg, hogy mennyire különböznek. Minél kisebb a távolság, annál közelebb van egymáshoz a két szöveg (tehát hasonlóbbak). Olyan esetekben, mikor a metrika egy szerkesztés alapú távolságot határoz meg, a következő matematikai átalakítással kaphatjuk meg a metrikához tartozó hasonlóságot: [9]

$$hasonlóság_{(S_1,S_2)} = 1 - \frac{távolság_{(S_1,S_2)}}{\max(|S_1|,|S_2|)}, \qquad (2.1.1)$$

ahol S_1 az első S_2 a második szöveg, $|S_1|$ az első $|S_2|$ a második szöveg karaktereinek száma (hossza).

Például, ha két szöveg közötti Levenshtein távolság (karakterenkénti szerkesztési lépések száma) 2, és a maximális hossz 10, akkor a $hasonlóság = 1 - \frac{2}{10} = 0{,}8$.

2.1.2.1 Normalizált Levenshtein

A Levenshtein távolság [10] esetén két szekvencia (jelen esetben a két szöveg karaktereinek sorozata) közötti távolság meghatározásához ki kell számítani az egykarakteres szerkesztések (beszúrások, törlések vagy helyettesítések) minimális számát, amelyek ahhoz szükségesek, hogy az egyik karaktersorozatból megkapjuk a másikat. Mivel ez a metrika egy távolság, így a kapott értékből ki kell számítani a hasonlóságot a **2.1.1.** képlet segítségével, amelyet már Normalizált Levenshtein távolságnak hívnak.

Az algoritmus futásideje (dinamikus programozással): $O(|S_1| * |S_2|)$.

2.1.2.2 Jaro Winkler

A Jaro-Winkler [11] metrika a Normalizált Levenshtein-hez hasonlóan egy karakteralapú szerkesztési távolság a két szekvencia között. A különbség a két metrika között, hogy két közeli karakter transzponálása kevésbé fontosnak tekinthető, mint két egymástól távol lévő karakter transzponálása. Továbbá ez a távolság bünteti azokat a hozzáadásokat vagy helyettesítéseket, amelyek nem fejezhetők ki transzpozícióként.

Az algoritmus futásideje: $O(|S_1| * |S_2|)$.

2.1.2.3 Koszinusz hasonlóság

A koszinusz hasonlóság [12] alapja a két nem nulla vektor közötti szög koszinusza. Kimente -1 és 1 közötti valós érték, ahol -1 a tökéletes ellentéteteket, 1 pedig a teljesen azonos szövegeket jelöli. Ehhez első lépésként a feldolgozandó szövegeket vektorizálni kell, majd az alábbiak szerint kiszámítani a hasonlóságot:

$$Sim_{cos}(S_1, S_2) = \frac{V_1 \cdot V_2}{|V_1||V_2|}, \tag{2.1.2}$$

ahol V_1 az első, V_2 a második szövegből kapott vektor, $|V_1|$ az első, $|V_2|$ a második szövegvektor hossza. A többi metrikával való összevetés érdekében normalizálni szokták az értéket 0 és 1 közé.

Az algoritmus futásideje: $O(|S_1| + |S_2|)$.

2.1.2.4 Jaccard-index

A Jaccard-index (más néven Jaccard hasonlósági együttható) [13] egy karakteralapú hasonlósági metrika, amely két (a karakterláncokban lévő karakterekből képzett) véges halmaz tagjait hasonlítja össze annak megállapítására, hogy mely tagok közösek és melyek különbözőek, amit a következőképpen kapunk:

$$Sim_{jaccard}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}, \tag{2.1.3}$$

Az algoritmus futásideje: $O(|S_1| + |S_2|)$.

2.1.2.5 Sorensen-Dice együttható

A Sorensen-Dice [14] együttható hasonló a Jaccard-indexhez, azzal a különbséggel, hogy a hasonlóságot az alábbiak szerint kell kiszámítani (a közös elemeket kétszeresen veszi figyelembe):

$$Sim_{sorensen}(S_1, S_2) = \frac{2|S_1 \cap S_2|}{|S_1| + |S_2|}, \tag{2.1.4}$$

Az algoritmus futásideje: $O(|S_1| + |S_2|)$.

2.1.2.6 Longest Common Subsequence

A leghosszabb közös részsorozat [15], mint távolság alapja, hogy megtaláljuk a két (vagy több) szekvenciában a leghosszabb közös részsorozatot úgy, hogy a karaktereknek nem kell egymást követő pozíciókat elfoglalniuk az eredeti szekvenciákon belül, vagyis kihagyhatunk karaktereket. Ez a távolság abban az esetben ad azonos eredményt a Levenshtein távolsággal, ha abban csak a beszúrás és a törlés művelet engedélyezett (a helyettesítés nem), vagy, ha a helyettesítés költsége a beszúrás vagy a törlés költségének kétszerese.

Az algoritmus futásideje: $O(|S_1| * |S_2|)$.

Mivel ezt a metrikát részben tartalmazza a Levenshtein távolság, illetve mivel megengedett a szöveg részeinek figyelmen kívül hagyása, így a hasonlóság vizsgálat szempontjából nem ideális, az egyes metrikák összevetésénél nem veszem figyelembe.

Természetesen az itt bemutatott metrikákon túl számtalan egyéb megvalósítás, illetve ezek módosított, javított változata is elérhető, azonban megkötésekkel, de ezen metrikák tekinthetők a legáltalánosabbaknak [7,16,17], így ezek fogják jelenteni az összehasonlítás alapját a saját módszeremmel.

2.1.2.7 Metrikák alkalmazhatósága forráskódok esetén

A bemutatott metrikák hasonlósági értékeit a forráskód részletek között a **2.1.1.** táblázat tartalmazza, ami alapján kijelenthető, hogy az eredmények nem tükröznek valódi hasonlóságot. Míg a szó (token) alapú metrikák

esetében a probléma az utasítások helytelen szétválasztása, addig a karakter (szerkesztési távolság) alapú módszerek esetében a karakterek egymásutánisága.

2.1.1. táblázat. *A különböző karakterlánc alapú metrikák hasonlósági értékei a* ***2.1.1.*** *ábrán látható forráskódok esetében.*

	A → B	**A → C**	**B → C**
Koszinus hasonlóság	0,4909	0,4220	0,3263
Jaccard index	0,4539	0,3245	0,3311
Sorensen-Dice együttható	0,6244	0,4900	0,4975
Normalizált-Levenshtein	0,5714	0,3412	0,3257
Jaro-Winkler	0,7612	0,5858	0,6472

A cél egy olyan hasonlósági metrika kidolgozása, amely nem igényli a forráskódok bonyolult előfeldolgozását, értelmezését, amellyel nem szükséges újabb absztrakciós szintet behozni, illetve használata nincsen megkötve valamely programozási nyelvre. Ezek alapján a tervezett hasonlósági metrika a forráskódot szöveges reprezentációjában fogja feldolgozni, azon karakter alapú hasonlóságot fog meghatározni.

2.1.3 Saját módszer bemutatása

Olyan megoldás kidolgozása a cél, amely képes kezelni a független utasítások, blokkok sorrendiségét, illetve az összehasonlítandó forráskódok hosszának különbségéből adódó eltéréseit. Erre jó megoldás lehet a leghosszabb közös részsorozat alapú módszer egy speciális változata, a leghosszabb közös részstring (Longest Common Substring – LCS) keresés, amely a leghosszabb egybefüggő részsorozatot keresi meg, vagyis nem megengedett a karakterek kihagyása. Ennek algoritmusát a **2.1.1** algoritmus mutatja be.

A **2.1.1.** algoritmus segítségével a forráskódban megtalálhatjuk azt a részsorozatot, amely a „leginkább megegyezik”. Miután ez megtörtént, kivágjuk mindkét forráskódból, majd újra elvégezzük a műveletet mindaddig, míg van közös részsorozat vagy valamelyik forráskód karakterei el nem fogynak. A megtalált részsorozatoknak meghatározzuk az arányát a teljes forráskódra nézve azok karaktereinek száma alapján, összegezzük őket így megkapva, hogy hány százalékban egyezik meg az egyik forráskód a másikkal, illetve fordítva. Mivel így két százalékos értéket is fogunk kapni, ezért szükséges lesz egy globális értéket is számítani ebből a két értékből.

2.1.1. algoritmus. *Leghosszabb közös részstring.*

Bemenet: S_1 – első szöveg, S_2 – második szöveg,
$|S_1|$ – első szöveg hossza, $|S_2|$ – második szöveg hossza
Kimenet: a leghosszabb közös részstring hossza

```
1:  function LCS(S1, S2, |S1|, |S2|)
2:      lCS ← 0
3:      for i ← 0 to |S1| do
4:          for j ← 0 to |S2| do
5:              k ← 0
6:              while (i + k) < |S1| ∧ (j + k) < |S2| ∧ S1^(i+k) = S2^(j+k) do
7:                  k ← k + 1
8:              end while
9:              lCS ← max(lCS, k)
10:         end for
11:     end for
12:     return lCS
13: end function
```

1. példa. Ha a leghosszabb közös részstring alapú módszerrel szeretnénk meghatározni a „MANHATTAN” és a „MAHATMA” szavak közötti hasonlóságot, akkor azt az alábbi lépések útján kapjuk meg:

1) A két szó leghosszabb közös egybefüggő részsorozata a „HAT”.
   - A HAT a MANHATTAN 3/9-ed része, vagyis 0,333-e.
   - A HAT a MAHATMA 3/7-ed része, vagyis 0,428-e.
2) Vágjuk ki a megtalált részsorozatot mindkét szóból, így kapva a MAN | TAN és a MA | MA szavakat, majd ezekben újból keressük meg a leghosszabb közös egybefüggő részsorozatot, ami az „MA” lesz.
   - A MA a MANHATTAN 2/9-ed része, vagyis 0.222-e.
   - A MA a MAHATMA 2/7-ed része, vagyis 0,285-e.

   (A végső eredmény számításánál kétszer kerül figyelembevételre az érték, mivel az eredeti szó is kétszer tartalmazta a részsorozatot. A későbbiekben, az algoritmus fejlesztésével azonban pontosabb értéket kaptam, amikor a szót a kisebb előfordulás szerint vettem figyelembe. A figyelembevételtől függetlenül a részsorozat minden előfordulása eltávolításra kerül a szavakból.)

3) Miután a `MA` szó is kivágásra került a `MANHATTAN` szóból az `NTAN` részlet marad, míg a `MAHATMA` szóban nem maradt további feldolgozatlan karakter, így a feldolgozás leáll.
4) Végül az eredmény egy százalékos érték lesz, amely a két karakterlánc közös karaktereinek arányát jelzi:
    - A `MANHATTAN` szó esetében ez 0,333 + 0,222 = 0,555.
    - A `MAHATMA` szó esetében ez 0,428 + 2 * 0,285 = 0,998 ≈ 1.

2. példa. Ha azonban az „`ÁLLAPOT`” és a „`LAPÁTOL`” anagrammáknak szeretnénk meghatározni a hasonlóságát, akkor az LCS alapú módszer mindkét szóra 1-et fog adni, az alábbi lépéseket követően:

1) A két szó leghosszabb közös egybefüggő részsorozata az „`AP`”.
    - Az „`LAP`” mindkét szó 3/7-ed része, vagyis 0,428-a.
2) Vágjuk ki a megtalált részsorozatot mindkét szóból, így kapva az `ÁL|OT` és a `ÁTOL` szavakat, majd ezekben újból keressük meg a leghosszabb közös egybefüggő részsorozatot. Mindegyik részsorozat 1-1 karakter hosszú lesz, melyeket akár egy lépésben is kezelhetünk.
    - Az „`Á`”, „`L`”, „`O`”, „`T`” egyszer fordul elő mindkét szóban, azok 1/7-ed részei, vagyis 0,142%-e.
3) Ezt követően egyik szóban sem maradt feldolgozatlan karakter, így az algoritmus leáll.
4) Végül az eredmény itt is egy százalékos érték lesz, amely a két karakterlánc közös karaktereinek aránya. Mivel minden karakter mindkét szóban feldolgozásra került, így a módszer 100%-os hasonlóságot adott a két szóra.

A 2. példa esetében felmerül a kérdés, hogy ez a mérőszám jó-e a gyakorlatban. A természetes nyelveknél feltehetően ennek sok értelme nem lenne, azonban a forráskódok esetében, mivel a független utasítások felcserélhetők, így feltételezhetően ezzel a megközelítéssel jobb hasonlósági értéket érhetünk el, mint a már létező szövegalapú hasonlósági metrikákkal.

A **2.1.1.** ábrán látható (**A**) és (**B**) forráskódok közötti közös részsorozatok alakulását a **2.1.2.** táblázat szemlélteti, illetve a **2.1.1.** ábrán bemutatott forráskódokra meghatározott LCS értékeket a **2.1.3.** táblázat tartalmazza.

2.1.2. táblázat. *Az „A” és „B” forráskódok közötti leghosszabb közös részsorozat, előfordulásuk és százalékos arányuk, valamint a két forráskód összegzett hasonlósági értéke.*

	A forráskód			B forráskód			Részsorozat
	O[1]	**%**	**TP[2]**	**O[1]**	**%**	**TP[2]**	
	2	0,095	0,190	1	0,110	0,110	'\r\n Console.WriteLine('
	1	0,091	0,091	1	0,105	0,105	'int[] ar)\r\n\{\r\n int s'
	1	0,075	0,075	1	0,087	0,087	'static void SumAvg('
	2	0,055	0,110	2	0,064	0,128	' = 0;\r\n for'
	1	0,055	0,055	1	0,064	0,064	';\r\n\r\n int p'
	1	0,047	0,047	1	0,055	0,055	')\r\n s'
	1	0,047	0,047	1	0,055	0,055	')\r\n p'
	1	0,019	0,019	1	0,022	0,022	');\r\n\}'
	2	0,015	0,030	2	0,018	0,018	'nt i'
	1	0,015	0,015	1	0,018	0,018	' += '
	1	0,015	0,015	1	0,018	0,018	' *= '
	1	0,007	0,007	2	0,009	0,018	'nt'
	2	0,007	0,014	2	0,009	0,018	' ('
	2	0,007	0,014	4	0,009	0,036	'ar'
Összegzett hasonlósági érték		S_{12P}=0,7420			S_{21P}=0,7752		

[1] részsorozat előfordulása; [2] összesített százalékos arány

2.1.3. táblázat. *A **2.1.1.** ábrán látható forráskódok minden kombinációjára kiszámított leghosszabb közös részstring módszerrel kapott hasonlósági értékek.*

		A → B	A → C	B → C
Longest Common Substring	S_{12P}	0,7420	0,4920	0,4770
	S_{21P}	0,7752	0,6587	0,6507

A hasonlósági értékek meghatározása után, mivel a két forráskódra két különböző értéket kapunk (a hosszuk eltérése miatt), a globális hasonlósági értéket kell kiszámítani. Globális érték esetén az a cél, hogy a nagyobb érték jobban figyelembe legyen véve, ezért súlyozzuk a százalékokat a forráskódok karaktereinek száma alapján az alábbiak szerint:

$$Score = \frac{\alpha(1-\beta)+\beta(1-\alpha)}{1-\beta+1-\alpha} \qquad (2.2.1)$$

ahol

- α a két hasonlósági érték minimuma: min (S_{12P}, S_{21P}),
- β a két hasonlósági érték maximuma: max (S_{12P}, S_{21P}).

A számított globális LCS alapú hasonlóság értékei a **2.1.1.** ábrán látható forráskódoknak a **2.1.4.** táblázatban kerültek feltüntetésre.

2.1.4. táblázat. *A különböző karakterlánc alapú metrikák hasonlósági értékei a* ***2.1.1.*** *ábrán látható forráskódok esetében, kiegészítve a leghosszabb közös részstring által adott globális hasonlósági értékekkel.*

	A → B	**A → C**	**B → C**
Koszinusz hasonlóság	0,4909	0,4220	0,3263
Jaccard index	0,4539	0,3245	0,3311
Sorensen-Dice együttható	0,6244	0,4900	0,4975
Normalizált-Levenshtein	0,5714	0,3412	0,3257
Jaro-Winkler	0,7612	0,5858	0,6472
Longest Common Substring	0,7597	0,5917	0,5812

A saját módszer eredményei jónak tekinthetők összevetve a vizsgált szöveg alapú hasonlósági metrikákkal, mivel a magas hasonlósági értékek azt igazolják, hogy a három forráskód ugyanazt a feladatot oldotta meg.

Azonban az összehasonlítás alapját képező forráskódok csak ugyanarra a feladatra adtak megoldást, így negatív példaként érdemes azt az esetet is megvizsgálni, amikor az összehasonlítandó forráskódok két eltérő feladat megoldásai, így működésük alapján sem tekinthetők hasonlónak. A kérdés eldöntésére a **2.1.2.** ábrán látható két C# programozási nyelven írt forráskódot választottam. A **2.1.2.** ábrán látható (**A**) forráskód a QuickSort algoritmus Linq megvalósítása, míg a (**B**) forráskód egy gráf szélességi bejárását valósítja meg és adja vissza a gráf csomópontjait a bejárás sorrendjében. A hasonlósági értékeket a **2.1.5.** táblázat tartalmazza. Az eredmények nem tekinthetők meglepőnek, figyelembevéve a programozási nyelvek felépítését, valamint az egyes metrikák működési elvét. Bár a saját módszerem által adott érték jelentősen meghaladja a legtöbb metrika által szolgáltatott értékeket, ez jól megválasztott, feladat függő küszöb értékkel kezelhető, figyelembe véve az összehasonlítandó forráskódok jellegét.

```
static IEnumerable<T> QSLinq<T>(IEnumerable<T> _items) where T : INumber<T>
{
    if (_items.Count() <= 1) return _items;
    var _pivot = _items.First();
    var _less = from _item in _items where _item < _pivot select _item;
    var _same = from _item in _items where _item == _pivot select _item;
    var _more = from _item in _items where _item > _pivot select _item;
    return QSLinq(_less).Concat(QSLinq(_same)).Concat(QSLinq(_more));
}
```

*(**A**) A QuickSort algoritmus Linq megvalósítása.*

```
static List<int> BFS(List<int>[] adj)
{
    bool[] visited = new bool[adj.Length];
    List<int> result = [];
    visited[0] = true;

    Queue<int> q = [];
    q.Enqueue(0);

    while (q.Count > 0)
    {
        int x = q.Dequeue();
        result.Add(x);
        foreach (int y in adj[x])
            if (!visited[y])
            {
                visited[y] = true;
                q.Enqueue(y);
            }
    }
    return result;
}
```

*(**B**) Metódus, ami egy gráf szélességi bejárását valósítja meg, és adja vissza a bejárt csomópontokat a bejárás sorrendjében.*

2.1.2. ábra. *Két eltérő feladat megoldásának forráskódja, amely példa a saját módszer alkalmazhatóságának vizsgálatára, mikor a hasonlósági értéknek alacsonynak kell lennie.*

2.1.5. táblázat. *A különböző karakterlánc alapú metrikák hasonlósági értékei a **2.1.2.** ábrán látható forráskódok esetében, kiegészítve a leghosszabb közös részstring által adott globális hasonlósági értékekkel.*

	A → B
Koszinusz hasonlóság	0,1376
Jaccard index	0,0948
Sorensen-Dice együttható	0,1732
Normalizált Levenshtein	0,1952
Jaro-Winkler	0,5940
Longest Common Substring	0,4984

2.1.3.1 LCS alapú hasonlóság futásideje

A leghosszabb közös részstring dinamikus programozással történő megtalálásának futási ideje $O(|S_1| * |S_2|)$, ahol $|S_1|$ az első, míg az $|S_2|$ a második karakterlánc hossza. A leghosszabb közös részstring megkeresését

és kivágását követően a lépést újra el kell végezni. [18] Ha k darab karakter kerül kivágásra az eredeti karakterláncokból, akkor a futásidő a legrosszabb esetben $O((|S_1| - k) * (|S_2| - k))$ lesz. Ezt a sorozatot addig ismétli az algoritmus, amíg nem talál egy másik részsorozatot, vagy amíg az egyik karakterlánc hossza el nem éri a 0-t. Ennek megfelelően a bemutatott metódus futásideje $O(|S_1| * |S_2| * x)$ lesz, ahol $|S_1|$, $|S_2|$ rendre az első és a második karakterláncok hossza, az x pedig az iterációk száma, melynek felső becslése $\min(|S_1|, |S_2|)$, vagyis a futásidő $O(|S_1| * |S_2| * \min(|S_1|, |S_2|))$.

2.1.4 A bemutatott módszer értékelése

A bemutatott módszerrel kapcsolatos eredmények bíztatók, az ismertetett metrikák jelentős részénél jobb eredményt mutatott a **2.1.1.** ábrán látható egyszerű példán. Azonban, mikor két merőben eltérő forráskód került összehasonlításra (**2.1.2.** ábra), az általa adott hasonlósági érték szintén magas volt, így szükségszerű több, komplexebb vizsgálatot is elvégezni, hogy teljes képet kapjunk a használhatóságáról. Erre, a mindenki számára elérhető HackerRank adatbázisát használtam, ahol a következő feladatokat választottam a teszteléshez:

- *Insertion Sort – Part 1* – A feladat a bemenetként kapott részben rendezetlen tömbben, az egyetlen rendezetlen elem rendezése beszúrásos rendezés segítségével úgy, hogy minden lépés (beszúrás, mozgatás) során a teljes tömböt meg kell jeleníteni a kimeneten. [19]
- *Insertion Sort – Part 2* – A feladat hasonló az *Insertion Sort – Part 1* feladathoz azzal a különbséggel, hogy a bemeneti tömb immár nem csak egy rendezetlen elemet tartalmazhat. [20]
- *Pairs* – A feladat a bemenetként megadott egész számokat tartalmazó tömbben megkeresni azokat az elempárokat, melyek különbsége megegyezik a szintén bemenetként megadott célértékkel. A feladat kimeneteként ezen elempárok számát kell megadni. [21]
- *Simple Array Sum* – A feladat a bemenetként megadott tömb elemeinek összegzése, amelynek eredményét a kimeneten kell megjeleníteni. [22]

Minden feladatból véletlenszerűen 100 darab olyan C# nyelven készült

megoldást töltöttem le anonimizálva, ahol a megoldás minden tesztesetet teljesített, így garantálva a forráskód helyes működését. A forráskódokon további átalakítások nem történtek, azok pontosan abban a formában kerültek összehasonlításra, amely formában a készítője leadta. A letöltött megoldások tulajdonságait a **2.1.6.** táblázat tartalmazza.

2.1.6. táblázat. *A HackerRank oldaláról letöltött feladatok megoldásainak tulajdonságai.*

Feladat neve	**Megoldás karaktereinek száma**			
	Legrövidebb	**Leghosszabb**	**Átlag**	**Medián**
Insertion Sort – Part 1	601	2 218	1 258,22	1 234
Insertion Sort – Part 2	792	2 268	1 240,32	1 165
Pair	533	4 348	1 370,95	1 266
Simple Array Sum	331	923	554,08	499

A különböző metrikák összehasonlításához elkészítettem az igazság mátrixokat, melyeket kiegészítettem kettő modellel. Az első modell minden esetben negatív választ adna, míg a másik modell véletlenszerűen produkálna eredményt. A két modell által kapott értékek, mint referencia értékek jelennek meg a következtetés levonása során.

Az igazság mátrixban az alábbiak szerint számoltam össze a kapott hasonlósági értékeket (ahol az összes azonos feladaton belüli összehasonlítások száma $P = 40\,000$, az összes különböző feladatokkal való összehasonlítások száma $N = 120\,000$):

- *Valós Pozitív (VP)* az eredmény, ha a hasonlósági érték nagyobb vagy egyenlő mint 0,98 (tetszés szerint választott küszöb érték), tehát az algoritmus azt feltételezte, hogy a két megoldás ugyanazt a feladatot oldotta meg, és ez valóban így is volt.
- *Valós Negatív (VN)* az eredmény, ha a hasonlósági érték kisebb, mint a küszöbérték és a megoldások valóban különböző feladathoz tartoznak.
- *Hamis Pozitív (HP)* az eredmény, ha a hasonlósági érték nagyobb vagy egyenlő, mint a küszöbérték, azonban a megoldások különböző feladathoz tartoznak.
- *Hamis Negatív (HN)* az eredmény, ha a hasonlósági érték kisebb, mint a küszöbérték, azonban a megoldások ugyanahhoz a feladathoz tartoznak.

A fentiek alapján a különböző metrikák igazság mátrixait a **2.1.7.** táblázat szemlélteti, illetve a **2.1.8.** táblázat foglalja össze az igazságmátrixból számított különböző értékeket. Az eredmények alapján elmondható, hogy habár a saját módszer jóval több hasonló megoldást talál a különböző feladatok megoldásait összehasonlítva (ami magyarázható a módszer működésével), az ugyanannak a feladatnak a megoldásainál mintegy négyszer több megoldást jelölt hasonlónak, mint a második „legjobbnak" tekinthető Jaro-Winkler hasonlóság. Az eredményeket összevetve a véletlenszerű választ adó modellel kijelenthető, hogy a hagyományos módszerek (így például a Koszinusz hasonlóság vagy a Jaccard index F1 pontszámai, amik 0,03 és 0,02 körül mozognak) rosszabbul teljesítenek egy egyszerű tippelésnél. Ugyanakkor a saját módszer pontossága nagyjából megegyezik a minden esetben negatív választ adó modellel, a módszer előnye a véletlenszerű modellel szemben az, hogy a hamis pozitív értékeket drasztikusan alacsonyan tartja ($HP_{LCS}=3690 < HP_{\text{véletlenszerű modell}}=60\,000$), így biztos alapokat ad a valós felismeréshez, még úgy is, hogy az a valódi pozitív arány rovására megy. Továbbá a saját módszer több mutatóban (pontosság, F1 pontszám, kritikus sikerességi mutató) is jobb eredményt ért el, mint a többi metrika, így nem csak bizonyos esetekben mondható jónak, hanem általánosságban is. Természetesen további kutatást igényel a túlságosan is magasnak mondható hamis pozitívnak jelölt megoldások száma, illetve a küszöbérték megválasztásának módja, esetleg feladatfüggő küszöbérték választásának lehetősége.

2.1.7. táblázat. *A vizsgált metrikák igazságmátrixai.*

Koszinusz hasonlóság		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **658**	*HN* **39 342**
	Különböző	*HP* **14**	*VN* **119 986**
Jaccard index		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **496**	*HN* **39 504**
	Különböző	*HP* **2**	*VN* **119 998**
Sorensen-Dice együttható		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **526**	*HN* **39 474**
	Különböző	*HP* **2**	*VN* **119 998**
Normalizált Levenshtein		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **450**	*HN* **39 550**
	Különböző	*HP* **0**	*VN* **120 000**
Jaro-Winkler		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **1 124**	*HN* **38 876**
	Különböző	*HP* **0**	*VN* **120 000**
Longest Common Substring		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **4 602**	*HN* **35 398**
	Különböző	*HP* **3 960**	*VN* **116 040**
Minden esetben negatív modell		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **0**	*HN* **40 000**
	Különböző	*HP* **0**	*VN* **120 000**
Véletlenszerű modell		Számított hasonlósági érték szerint	
		>= 0,98	< 0,98
Megoldott feladat szerint	Azonos	*VP* **20 000**	*HN* **20 000**
	Különböző	*HP* **60 000**	*VN* **60 000**

2.1.8. táblázat. *A **2.1.7.** táblázat igazságmátrixaiból számított értékek.*

	Valódi pozitív arány (VPR) $\frac{VP}{P}$	**Pozitív prediktív érték** (PPE) $\frac{VP}{VP+HP}$	**Pontosság** $\frac{VP+VN}{P+N}$	**F1 pontszám** $\frac{2*PPE*VPR}{PPE+VPR}$	**Kritikus sikerességi mutató** $\frac{VP}{VP+HN+HP}$
Koszinusz hasonlóság	0,01645	0,97916	0,75402	0,03235	0,01644
Jaccard index	0,01240	0,99598	0,75308	0,02449	0,01239
Sorensen-Dice együttható	0,01315	0,99621	0,75327	0,02595	0,01314
Normalizált Levenshtein	0,01125	1,00000	0,75281	0,02250	0,01125
Jaro-Winkler	0,02810	1,00000	0,75702	0,05466	0,02810
Longest Common Substring	0,11505	0,53749	0,75401	0,18953	0,10468
Minden esetben negatív modell	0	-	0,75000	-	0
Véletlenszerű modell	0,50000	0,25000	0,50000	0,33333	0,20000

2.1.5 Konklúzió

A kidolgozott módszerről az eredmények alapján elmondható, hogy a gyakran használt szövegalapú hasonlósági metrikáknál jobb eredményt sikerült elérni a forráskódok tekintetében. Jegyezzük meg azonban, hogy a forráskódok előfeldolgozásával (felesleges kommentek, szóközök, és egyéb elemek eltávolításával, egységes formázás alkalmazásával stb.) tovább javíthatók az eredmények. A hasonlóságot a két hasonlósági értékből képzett származtatott értékkel határoztam meg, azonban további kutatást igényel ezek pontos, feladat szerinti definiálása (például maximum, harmonikus közép stb.). A módszer hiányossága, hogy ha az egyik

forráskód részleteiben, de teljes egészében megtalálható egy másik forráskódban, akkor a hasonlósági érték 1 lesz, mely szintén félrevezető lehet, ugyanis ez nem feltétlenül jelenti, hogy azonosak, inkább csak azt, hogy azonos elemeket (kulcsszavakat, elnevezéseket), struktúrákat használtak mindkét esetben. Ehhez szorosan kapcsolódik a hasonlósági határérték megválasztása, mely a valódi feladatmegoldások esetében 0,98-nak lett megválasztva. Ennek helyes meghatározása kulcsfontosságú a hasonlóságok eldöntésénél, mely a feladatra adható megoldások ismeretének hiányában nem lehetséges.

A fent említett problémák ellenére a módszer jól használható, ami a valós példákon keresztül is látható, azonban a téma további kutatást igényel, a különleges eseteket is megvizsgálva, illetve az algoritmus párhuzamosítási lehetőségeit is számba véve. Meggondolandó még a karakter alapú feldolgozás leváltása token alapú megközelítésre, ami jobban reprezentálná a forráskódok közötti hasonlóságot, azonban ez a forráskódok további előfeldolgozását tenné szükségessé.

2.1.6 Tézis kimondása

1.1. tézis – Újfajta mérőszámot dolgoztam ki forráskódok szövegalapú összehasonlítására, amely a leghosszabb közös részstring (Longest Common Substring - LCS) keresés iteratív alkalmazásán alapul. Ezen mérőszám és egy megfelelően választott határérték segítségével egy új módszert dolgoztam ki, amelyik két forráskódról képes megállapítani, hogy azok ugyanazt a feladatot oldják-e meg. A módszert 4 különböző feladatot megoldó, 400 darab, ember által készített forráskód egymással való összehasonlításával validáltam, amely alapján igazolni tudtam, hogy az jelentősen magasabb F1 score értéket ért el, mint a meglévő hasonló célú módszerek. [S1]

2.2 LCS algoritmus adatpárhuzamos modellje

2.2.1 Bevezetés

Az **1.1.** tézisben is hivatkozott, két szöveg leghosszabb közös részkarakterlánca az az egybefüggő karaktersorozat, amely mindkét szövegben megjelenik, és ezek közül a leghosszabb. A módszert széles körben alkalmazzák számos szöveghasonlósági mérés során, jellemzően egymást követően többször is ugyanazokon az adatokon. A probléma megoldására az idők során már több módszert is kidolgoztak, de ezek többnyire nagyon idő- és memóriaigényes eljárásokon alapulnak. A következőkben részletezett újszerű adatpárhuzamos modell jelentősége az általános párhuzamosság mellett a GPU-n való megvalósíthatóság, melyet a kísérletek eredményei is alátámasztanak.

A leghosszabb közös részkarakterlánc-problémában az a cél, hogy megtaláljuk a leghosszabb közös és egybefüggő karaktersorozatot ugyanabban a sorrendben két megadott karakterláncban. Legyen S_1 és S_2 a karakterláncok, $|S_1|$ és $|S_2|$ pedig ezek hossza. Egy adott S_1 karakterlánc i-edik karaktere S_1^i, ahol i értéke 1 és $|S_1|$ között van (ugyanez a jelölés kerül használatra S_2 esetén is). [23, 24]

Az eredeti probléma általánosításaként meg lehet találni az összes létező leghosszabb közös sorozatot. Ebben az esetben egy ciklust kell használni, hogy megkeressük a ténylegesen leghosszabb részkarakterláncot, amit kivág a karakterláncokból. Ezeket a lépéseket ismételjük mindaddig, amíg már nem fordul elő újabb közös részsorozat. A módszer eredménye a szöveges adatok hasonlóságának ellenőrzésére használható. A valós alkalmazások lehetnek hosszú dokumentumok [25] vagy forráskódok [26] összehasonlítása, plágiumellenőrzése.

2.2.2 A szakirodalomban elterjedt módszerek

A problémára már többféle megoldás is létezik, de legtöbbjük nagy számítási- és memóriaigényű. Ezek általában hagyományos szekvenciális algoritmusokon alapulnak, így érdemes megfontolni egy újszerű adatpárhuzamos algoritmus tervezését és megvalósítását [27], mely grafikus gyorsítók segítségével dolgozik, ezzel növelve a hatékonyságot [28,29].

2.2.2.1 Naiv megvalósítás

A legegyszerűbb megvalósítás, amikor az első karakterlánc összes részkarakterláncát egyenként figyelembe veszi és ellenőrzi, hogy a második karakterláncban is szerepel-e. Ez az algoritmus három egymásba ágyazott cikluson alapul, amint az a **2.2.1.** algoritmusban is látható. Az első ciklus S_1 karakterein iterál végig az i változó használatával, a második az S_2 karakterein a j változóval, a harmadik pedig az S_1^i és S_2^j egyező karakterek számát számolja a k változó használatával. Ennek eléréséhez a $k = 0$-val kezdődik és növelésre kerül, amikor $S_1^{i+k} = S_2^{j+k}$ és $i + k \leq |S_1|$ és $j + k \leq |S_2|$. A harmadik ciklus végén a k tartalmazza az S_1^i és S_2^j helyekről induló leghosszabb egybefüggő karakterláncának hosszát, amelyet minden i és j iterációhoz ki kell számítani. Végül ezen értékek maximuma lesz a végeredmény.

Az algoritmus futásideje $O(n^3)$, ahol az $n = \max(|S_1|, |S_2|)$, mivel az első ciklus futásideje $O(|S_1|)$, a középső belső ciklus futásideje $O(|S_2|)$, a legbelső ciklus futásideje pedig $O(\max(|S_1|, |S_2|))$. Ez a lépésszám nagy szövegek esetén elfogadhatatlan, a futásidő pár ezer karakteres szövegek esetén is percekben mérhető. Tagadhatatlan előnye azonban az algoritmusnak a minimális memóriaigénye (4 egész típusú lokális változó).

2.2.1. algoritmus. *Naiv megvalósítás.*

Bemenet: S_1 – első szöveg, S_2 – második szöveg,
$|S_1|$ – első szöveg hossza, $|S_2|$ – második szöveg hossza
Kimenet: a leghosszabb közös részsorozat hossza

```
1:  function NAIVELCS(S1, S2, |S1|, |S2|)
2:      lCS ← 0
3:      for i ← 0 to |S1| do
4:          for j ← 0 to |S2| do
5:              k ← 0
6:              while (i + k) < |S1| ∧ (j + k) < |S2| ∧ S1^(i+k) = S2^(j+k) do
7:                  k ← k + 1
8:              end while
9:              lCS ← max(lcs,k)
10:         end for
11:     end for
12:     return lCS
13: end function
```

2.2.2.2 Rekurzív megvalósítás

A megoldás egy másik megközelítése a leghosszabb közös utótagokon (suffix) alapuló oszd meg és uralkodj elvet alapul vevő rekurzív megvalósítás használata. A rekurzió alapját a következő függvény fogja jelenteni:

$$r(S_1, S_2, |S_1|, |S_2|) \tag{2.2.1}$$

ahol az r a leghosszabb közös utótagja (suffix) az S_1 és S_2 stringben $|S_1|$ illetve $|S_2|$ karakternél.

A következő három szabály lefedi az összes előforduló esetet, és mindegyik egy egyszerűbb esetre redukálja a problémát:

- Triviális megoldásként, ha valamelyik karakterlánc üres, az eredmény 0.

  $$r(S_1, S_2, |S_1|, |S_1|) = 0 \tag{2.2.2}$$

  ahol $|S_1| = 0 \vee |S_2| = 0$.

- Ha a karakterláncok utolsó karakterei nem egyeznek ($S_1^{|S_1|} \neq S_2^{|S_2|}$), akkor az eredmény a maradék két karakterláncra számított eredmények maximuma.

  $$\max\left(r(S_1, S_2, |S_1| - 1, |S_2|), r(S_1, S_2, |S_1|, |S_2| - 1)\right) \tag{2.2.3}$$

  ahol $S_1^{|S_1|} \neq S_2^{|S_2|}$.

- Ha az utolsó karakterek egyeznek ($S_1^{|S_1|} = S_2^{|S_2|}$), akkor az eredmény 1 + a leghosszabb közös utótag hossza az utolsó karakterek kihagyásával kapott két karakterláncban. Ez utóbbi könnyen kiszámítható rekurzív hívással:

  $$r(S_1, S_2, |S_1|, |S_2|) = r(S_1, S_2, |S_1| - 1, |S_2| - 1) + 1 \tag{2.2.4}$$

  ahol $S_1^{|S_1|} = S_2^{|S_2|}$.

Ezek alapján a rekurzív megvalósítást a **2.2.2.** algoritmus mutatja be.

2.2.2. algoritmus. *Rekurzív megvalósítás.*

Bemenet: S_1 – első szöveg, S_2 – második szöveg,
$|S_1|$ – első szöveg hossza, $|S_2|$ – második szöveg hossza
Kimenet: a leghosszabb közös részsorozat hossza

```
1:  function r(S1, S2, |S1|, |S2|)
2:      if |S1| = 0 ∨ |S2| = 0 then
3:          return 0
4:      else
5:          if S1^|S1| ≠ S2^|S2| then
6:              return max (r(S1, S2, |S1|−1, |S2|),
                            r(S1, S2, |S1|, |S2| − 1))
7:          else
8:              return r(S1, S2, |S1| − 1, |S2| − 1) + 1
9:          end if
10:     end if
12: end function
```

2.2.2.3 Dinamikus programozás alapú megvalósítás

A leghosszabb közös részkarakterlánc probléma dinamikus programozáson alapuló megoldása minden számítástechnikai tananyag általános része. [30] Az ötlet, hogy mindkét karakterlánc összes részkarakterláncához megkeressük a leghosszabb közös utótag hosszát és eltároljuk ezeket a hosszokat egy táblázatban. [31]

A dinamikus programozási feladatoknál alapvető lépés a probléma optimális részfeladat tulajdonságának bizonyítása. Ezt követően egy táblázatot kell kitölteni a megfelelő sorrendben egy adott függvény segítségével.

A megoldás folyamata elvileg ugyanazon a koncepción alapul, mint a rekurzív. Itt azonban nincs szükség a rekurzív függvényhívásokra, ugyanis ezesetben a függvény segítségével egy (D) táblázatot töltünk ki. Egy másik fontos különbség, hogy a részeredményeket alulról felfelé gyűjtjük (ellentétben a rekurzív esettel, amikor felülről lefelé irányul a megközelítés). A triviális esetek kezelésére a táblázatot további sorral (0. sor) és oszloppal (0. oszlop) kell bővíteni.

A **2.2.3.** algoritmus a dinamikus programozáson alapuló megoldást mutatja. A módszer hátránya a nagy memóriaigény, mivel $(|S_1| + 1) * (|S_2| + 1)$ méretű egész típusú tömbre van szükség a (rész)eredmények tárolására.

2.2.3. algoritmus. *Dinamikus programozás alapú megvalósítás.*

```
Bemenet: S1 – első szöveg, S2 – második szöveg,
         |S1| – első szöveg hossza, |S2| – második szöveg hossza
Kimenet: a leghosszabb közös részsorozat hossza
 1: function r(S1, S2, |S1|, |S2|)
 2:     D ← [|S1| + 1, |S2| + 1]                    ▷ 0 alapú indexelés
 3:     lCS ← 0
 4:     for i ← 0 to |S1| do
 5:         D[i, 0] ← 0
 6:     end for
 7:     for j ← 0 to |S2| do
 8:         D[0, j] ← 0
 9:     end for
10:     for i ← 1 to |S1| do
11:         for j ← 1 to |S2| do
12:             if S1^i = S2^j then
13:                 D[i, j] ← D[i-1, j-1] + 1
14:                 lCS ← max(lCS,D[i, j])
15:             else
16:                 D[i, j] ← max(D[i – 1, j], D[i, j – 1])
17:             end if
18:         end for
19:     end for
20:     return lCS
21: end function
```

2.2.2.4 Szövegek összehasonlítása LCS alapon

A szövegösszehasonlítás alapötlete, hogy összegezzük a közös karakterláncok hosszát. Vagyis vegyük a két forrás összes leghosszabb közös (összefüggő) részkarakterláncát, határozzuk meg mindkét karakterláncra az arányukat (a legkisebb előfordulást figyelembe véve), majd vágjuk ki őket. Ismételjük addig ezt a lépést, amíg még találunk közös halmazt. Miután megtaláltuk az összes részstring sorozatot, arányukat összegezve megkapjuk az egyik karakterlánc százalékos arányát a másikhoz viszonyítva, és fordítva.

Ez a részkarakterlánc-vágás nem vezethet új, összefűzött karakterlánc létrehozásához, mivel ez olyan közös sorozatokat eredményezhet, amelyek eredetileg nem szerepeltek a bemeneti szövegben. Például az „AYABCXC” és az „YXABCC” karakterláncok esetében, ha az „ABC” kivágása után összefűznénk az újonnan létrejövő karakterláncokat, akkor az „AYXC” és az

„YXC” karakterláncokat kapnánk, amelyek leghosszabb közös része az „YXC”, mely nem szerepel az eredeti szövegben. Ennek elkerülése érdekében vagy külön ellenőrizzük a karakterlánc új részeit, vagy összefűzzük az újonnan létrejövő karakterláncokat valamely nem szöveges elválasztó karakterekkel.

Ezek alapján a hasonlóság mérésének folyamata a következő:

- A S_1 és S_2 leghosszabb közös részkarakterláncának lekérése k hosszúsággal
- A hasonlósági arányok kiszámítása és összegyűjtése: $\frac{k}{|S_1|}$ és $\frac{k}{|S_2|}$.
- A megtalált leghosszabb közös részkarakterlánc lecserélése nem szöveges karakterekre.
- A folyamat ismétlése amíg $k \neq 0$.

Ezek alapján kiszámítható a globális hasonlósági értéke a két karaktersorozatnak az alábbi képlet segítéségével:

$$Score = \frac{\alpha(1-\beta)+\beta(1-\alpha)}{1-\beta+1-\alpha} \tag{2.2.5}$$

ahol

- α a két hasonlósági érték minimuma: min (S_{12P}, S_{21P}),
- β a két hasonlósági érték maximuma: max (S_{12P}, S_{21P}).

2.2.3 Saját módszer bemutatása

2.2.3.1 Intuíció

Bár a naiv megoldás nagyon számításigényes, van egy nagy előnye a többihez képest: jól párhuzamosítható. Ezenkívül könnyen implementálható adatpárhuzamos módon, lehetővé téve a grafikus gyorsítókon való megvalósítást. Ezek a hardvereszközök több ezer számítási egységet tartalmaznak, amelyek elérhetővé teszik a számítások felgyorsítását.

Ez a több ezer számítási egység lehetővé teszi a következő adatpárhuzamos modell tervezését és megvalósítását:

- Elindítunk $|S_1| * |S_2|$ számú szálat.
- A szálak indexei legyenek $1..|S_1|$, $1..|S_2|$ (a CUDA terminológiája szerint ezek az indexek $threadIdx.x$ és $threadIdx.y$, bár a tényleges implementációban 0-val kezdődő indexelésre lesz szükség).
- Minden szál kiszámítja a leghosszabb részkarakterlánc hosszát $S_1[threadIdx.x]$ és $S_2[threadIdx.y]$ értéktől kezdve.
- Egy atomi művelet segítségével a szálak kiválasztják a maximális hosszúságot.
- A leghosszabb közös részsorozatot megtaláló szálak a paramétereiket eltárolják egy ideiglenes tömbben.
- Ezt egy minden eszközre és blokkra vonatkozó szinkronizáció követi, aminek hatására az ideiglenes tömb tárolni fogja az összes leghosszabb közös részsorozat paramétereit.
- Egy GPU→CPU memóriamásolás segítségével az ideiglenes tömb minden adata átkerül a CPU központi memória tartományába.
- Az algoritmus működése ezt követően már megegyezik az eredeti változattal, a CPU tovább tud dolgozni a megtalált értékekkel.

Egy adott szál lépéseinek maximális száma $\max(|S_1|, |S_2|)$. Ideális esetben, ha az összes szál párhuzamosan futhat, az algoritmus komplexitása $O(n)$, ahol $n = \max(|S_1|, |S_2|)$, amely két nagyságrenddel jobb, mint a naiv implementáció, és lényegesen jobb, mint a dinamikus programozási megoldás.

2.2.3.2 Feladatok felvágása

Habár egy modern GPU több ezer feldolgozó egységet tartalmaz, nem garantált, hogy annyi szálat tud majd párhuzamosan elindítani, amennyi szükséges ($|S_1| * |S_2|$). Ezért további finomításra van szükség ahhoz, hogy a feladatot kisebb önálló részfeladatokra lehessen bontani. A CUDA terminológiát használva ez a független blokkok meghatározását jelenti, amelyek párhuzamosan, de szekvenciálisan is futtathatók a GPU képességei alapján.

Létezik azonban egy hardveres korlát, amely meghatározza az egy blokkban elindítható szálak maximális számát. Ez a kezdeti CUDA kompatibilis NVIDIA GPU-kon 512 volt, a modern GPU-kon (ahol a compute capability értéke legalább 2.0) pedig már 1024. A megoldás megtervezése során feltételeztem, hogy annak implementációjakor az egy blokkban futtatható szálak száma 1024 lesz.

A blokkok száma is korlátozott, ami maximum 65535 lehet. Ennek értelmében az újszerű adatpárhuzamos részstring kereső kernel a következőkre épül:

- Ha $|S_1| > |S_2|$, akkor $S_1 \leftrightarrow S_2$, $|S_1| \leftrightarrow |S_2|$.
- A blokkok száma $|S_2|$. Minden szál elkezdi ellenőrizni az S_2 karakterláncot a $blockIdx.x$ (a blokk indexe) pozícióban.
- A szálak száma $\min(|S_1|, 1024)$. Minden szál egy számlálós ciklust indít i indexszel 0 és $\left\lfloor \frac{|S_1|-1}{1024} \right\rfloor$ között. A szál minden $threadIdx.x + i * 1024$ pozícióban ellenőrzi az S_1 karakterláncot.

Feltételezzük, hogy egyik karakterlánc sem hosszabb 65535 karakternél. Ha lehetséges, ki kell bővíteni a modellt egy másik ciklussal, hogy az S_2 összes karakterét több ciklusban feldolgozzuk.

2.2.3.3 Ideiglenes eredmények begyűjtése

A kernelek több blokkban történő elindítása számos mellékhatással jár. A fő hátrány az, hogy már nem garantált, hogy az adatpárhuzamos szálak ténylegesen párhuzamosan lesznek végrehajtva. A grafikus hardver erőforrásai (és a bemeneti szövegek mérete) alapján elképzelhető, hogy a GPU csak akkor tud egy adott blokkot elindítani, ha a korábban elindított blokkok egy része befejeződik. Jól megjósolható és elvárható viselkedés, de bizonyos műveleteket lehetetlenné tesz. Nem lehet minden szálra egyetlen atomi műveletet használni a közös részstring maximális hosszának meghatározásához, mert előfordulhat, hogy az első blokkban lévő szálak már befejeződtek, amikor egy másik blokk másik szála megtalálja a leghosszabb közös részstringet.

Ezért a szálak részeredményeinek tárolására az ideiglenes tárolást használják. A kernel végén az összes ténylegesen futó szál végrehajtja az

atomic max műveletet egymással versengve, hogy megtalálják a legnagyobb számot, és a nyertesek ezt az értéket és a koordinátáikat tárolják az ideiglenes tárolóban. Ez azonban nem a végeredmény, mert a további blokkban lévő szálak hosszabb részstringeket találhatnak, így ezeket is az ideiglenes tárolóban tárolják.

Ez az ideiglenes tárolás tárolja az összes blokk "hullám" eredményét és néhány nem megfelelő értéket az elkerülhetetlen versenyfeltételek miatt. A kernel végrehajtásának végén egy további kernelindítás szükséges, hogy a legjobb eredményeket gyűjtsük össze ebből az ideiglenes tárolóból. Ez a probléma adatpárhuzamos módon is kezelhető, egyszerre több szál használatával.

2.2.3.4 Részsorozatok eltávolítása

Az általános elképzelés szerint mindkét bemeneti karakterláncból ki kell vágni az összes közös részstringet. A CPU-ban általában egy egyszerű szekvenciális algoritmus valósítja meg, hogy az összes karaktert előrébb másolja a közös részstring előtt (vagyis kivágjuk a közös részstringet). A GPU-ban ez a memóriaigényes eljárás adatpárhuzamosan is végrehajtható. A művelethez célszerű elindítani a hardver által támogatott maximális számú szálat, ahol minden szál csak egy karaktert mozgat előre. A feldolgozó egységek nagy számának köszönhetően ez a párhuzamosítás jelentősen felgyorsíthatja a működést.

2.2.4 A bemutatott módszer értékelése

Az új módszer pontosságának és gyorsaságának ellenőrzésére számos mérés került végrehajtásra. Jogos elvárás volt, hogy a GPU modell által adott eredmények megegyezzenek az eredeti megvalósítással. Egy másik elvárás volt, hogy a GPU kódja lényegesen gyorsabb legyen, mint a CPU-n lévő megvalósítás.

A mérések során többfajta karakterlánc került felhasználásra különböző kategóriákból: egyedi szavak, mondatok, rövid forráskódok és hosszú forráskódok. Ezek összehasonlítása a jól ismert dinamikus programozás alapú módszerrel és az adatpárhuzamos modell CUDA implementációjával történt.

A mérések során használt konfiguráció:

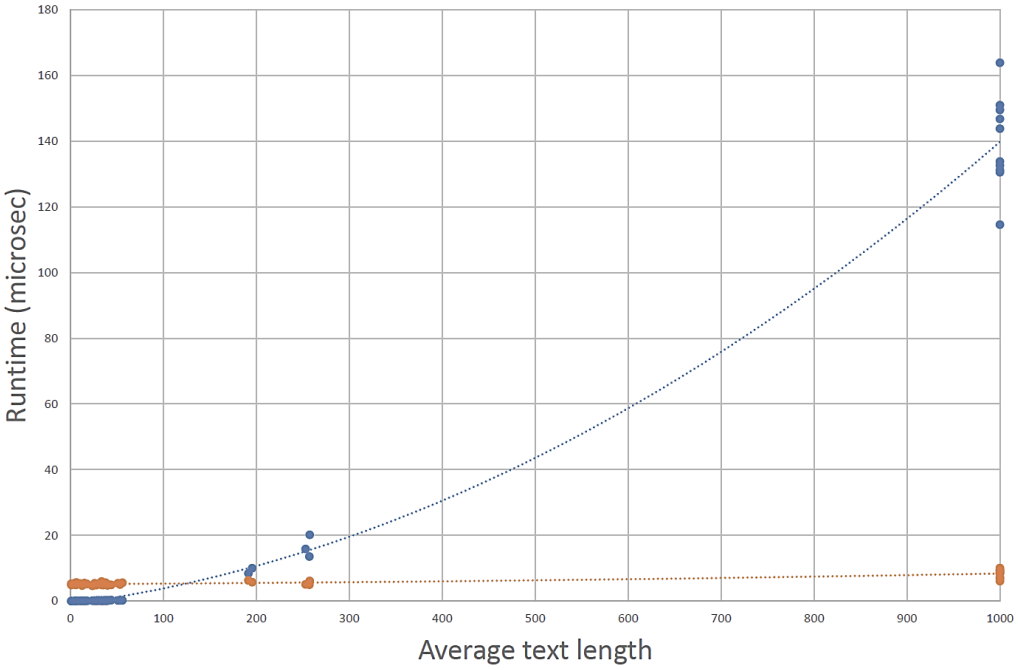
- CPU konfiguráció:
  - Processzor: Intel® Core™ i7-10700k
  - Architektúra: Comet Lake
  - Magok száma: 8
  - Memória: 32GB DDR3
- GPU konfiguráció:
  - Grafikai gyorsító: NVIDIA Quadro RTX6000
  - Architektúra: Turing
  - Shader-ek száma: 4608
  - Memória: 24GB GDDR6

A **2.2.1.** táblázat a különböző szövegek összehasonlításának részletes eredményeit mutatja. Amint látható, a hasonlósági pontszám általában azonos vagy nagyon hasonló. A futásidő kis elemszám esetében még a CPU-n volt alacsonyabb, a szövegek méretének növelésével azonban már egyértelművé vált a GPU modell előnye.

2.2.1. táblázat. *CPU és GPU implementáció összehasonlításának eredményei.*

$\|S_1\|$	$\|S_2\|$	**Hasonlóság CPU**	**Hasonlóság GPU**	**Futásidő CPU(µs)**	**Futásidő GPU(µs)**
1	1	0.000	0.000	0.003	4.830
6	7	0.625	0.625	0.018	5.635
10	10	0.600	0.600	0.031	5.274
20	55	0.753	0.753	0.124	5.147
196	195	0.778	0.775	10.011	5.703
187	*319*	*0.605*	*0.612*	*15.847*	*5.050*
1000	*1000*	*0.858*	*0.855*	*133.776*	*8.476*

A **2.2.1.** ábrán a különböző hosszúságú szövegek átlagos futásidejét mutatja. A pontok egy adott vizsgálat futásidejét jelentik a megadott átlagos szöveghosszal ($\frac{|S_1|+|S_2|}{2}$). Ahogy a trendvonalakból is látszik, a GPU jobban teljesít a hosszú stringek esetében, mivel ezekben az esetekben az összes feldolgozó egységet képes kihasználni.



2.2.1. ábra. *A CPU és a GPU futásidejének összehasonlítása a bemeneti szövegek átlagos hossza alapján. A kék pontok a CPU, a piros pontok a GPU méréseit jelzik.*

2.2.5 Konklúzió

Amint az az eredményekből látható, gyakorlatban releváns feladatméretek esetében a GPU kód látványosan gyorsabb, mint a CPU. Ahogy az várható volt, a gyorsulás a bemeneti karakterláncok hosszával nő. Rövid szövegek esetében nem érdemes GPU-t használni, a nagy memóriaátviteli és kernelindítási igény miatt. De a körülbelül ~130 karakternél hosszabb szövegek esetén az adatpárhuzamos algoritmus gyorsabbá válik, és lényegesen jobb futásidejű eredmény érhető el a valós fájl összehasonlítási példákban.

A legtöbb GPU-alapú alkalmazás esetében jelentős probléma a memóriaátvitelek száma. Ez ebben az esetben is igaz, a forrásszövegek grafikus kártyára másolása meglehetősen időigényes tevékenység. További kutatásként a következő kérdések megoldására van szükség:

- Általában nemcsak két, hanem n számú szöveg összehasonlítására van szükség, amihez $n * (n - 1)$ összehasonlítás szükséges. Ebben az esetben elegendő a forrásszövegeket egyszer átmásolni a GPU-ra, így ennek a memóriaátvitelnek a hatása a teljes futási időre kisebb lesz.

- Többszörös összehasonlítás esetén lehetőség van egy összehasonlító kernel elindítására a GPU-ban; közben a következő összehasonlítás memóriaátvitele is megtörténhet. Ezzel az átfedő módszerrel lehetőség van az adatmozgatások elrejtésére.

2.2.6 Tézis kimondása

1.2. tézis – Kidolgoztam egy újszerű adatpárhuzamos módszert szövegek hasonlóságának mérésére. A módszer a leghosszabb közös részstring alapú összehasonlító módszer iteratív megvalósításán alapul, az abban található erőforrásigényes összehasonlításokat, illetve a megtalált részstringek eltávolítását azonban jóval hatékonyabban, adatpárhuzamos módon végzi el. A dinamikus programozás alapú szekvenciális és a GPU-n implementált adatpárhuzamos módszerek összehasonlítása alapján megállapítottam, hogy a nagyon rövid szövegek kivételével az általam kidolgozott módszer jelentősen kisebb futásidőt igényel, miközben azonos pontosságot ér el. [S2]

2.3 Egymásba ágyazott ciklusok párhuzamosítása

2.3.1 Bevezetés

Egy adathalmaz duplikált elemeinek megkeresésére többféle lehetőség is adódik, de azokban az esetekben, amikor a halmaz mérete drasztikusan nagy, az egyszerű szekvenciális megoldások komoly futási korlátokkal rendelkeznek. Ilyen esetekben tanácsos párhuzamosítani ezt a folyamatot a gyorsabb végrehajtás érdekében.

A duplikátumok megtalálásának legáltalánosabb módja a nyers erő módszere, amely minden elemet összehasonlít az adathalmaz minden más elemével. Ennek a megoldásnak a futásideje $O(n^2)$. Tekintettel arra, hogy a futásidő ilyen magas, illetve, hogy felesleges összehasonlításokat is eredményez, így a valóságban nagyon ritkán használják. [32] A nyers erő módszerének egy továbbfejlesztett változata kihasználja az elemeken végzett művelet reflexív tulajdonságát (mivel minden elem önmagával azonos), illetve szimmetrikusságát. Az elemek szimmetriája azt jelenti, hogy ha egy elem azonos egy másik elemmel, akkor fordítva (a relációban az elemek felcserélésével) is azonosak lesznek. Halmazok esetében a harmadik tulajdonság a tranzitivitás, mely egy általánosabb megoldás érdekében nem megengedett, így megakadályozva a láncszerűséget. Amennyiben engednénk a tranzitivitást úgy olyan elemek is egy osztályba kerülnének, melyek között nincs közvetlen hasonlósági kapcsolat, csupán azáltal, hogy a láncolatban szereplő elempároknak van közös metszete, mely a lánc két végén lévő elemekre már nem áll fenn. Ezen szabályok alapján egy nem üres $H \neq \emptyset$ halmazt és egy reflexív, szimmetrikus, nem tranzitív R relációt kapunk:

$$\forall a \in H: (aRa)$$

$$\forall a, b \in H: (aRb \Leftrightarrow bRa)$$

$$\exists a, b, c \in H: (aRb \wedge bRc) \not\Rightarrow aRc$$

Tegyük fel, hogy kompatibilitási osztályokat akarunk létrehozni az adatkészlet elemei között úgy, hogy az egyező elemeket (duplikátumokat) egy osztálynak tekintjük. Ahhoz, hogy ezt egy mátrixként tudjuk ábrázolni, és szemléltetni az összehasonlításokat, definiáljuk az S szekvenciát, amelynek elemei megegyeznek az H halmaz elemeivel, immár értelmezve a

sorrendiséget. Így hivatkozhatunk az S szekvencia i-edik és j-edik elemére, ami alapján a mátrix egy cellájának értéke: $matrix[i,j] = R(S[i], S[j])$. Ezek alapján a **2.3.1.** ábrái szerint lehet szemléltetni a mátrixokat, ahol a sárgával jelölt cellák jelentik az összehasonlítandó elemeket. Ha az R relációt figyelmen kívül hagyjuk, vagyis a nyers erő algoritmust használjuk a feldolgozásban, akkor egy 20 elemből álló halmaz esetén 400 összehasonlítás szükséges (**2.3.1.** (**B**) ábra). Ha figyelembe vesszük a reláció reflexivitását, akkor 380-ra csökkenthető az összehasonlítások száma (**2.3.1.** (**C**) ábra), mivel a főátló elemei elhagyhatók, az elemek önmagukkal azonosak lesznek. Ha a szimmetrikus tulajdonság a reflexivitás mellett szerepel, az összehasonlítások száma 190-re csökken (**2.3.1.** (**D**) ábra), mivel a tulajdonság garantálja, hogy két elem az összehasonlítás szempontjából felcserélhető. A tranzitivitás hiánya miatt a mátrix alsó háromszögén kell csak áthaladni, hogy megtaláljuk az összes azonos elemet az egyes elemekhez.

Az említett halmaz tulajdonságok figyelembevételével a halmaz méretétől (N) függően $\frac{N^2-N}{2}$ összehasonlításra lesz szükség, ami nem csökkenthető tovább (pontosabban nem csökkenthető tovább az adathalmazra vonatkozó további információk nélkül). Például, ha az adatkészlet 10 000 elemet tartalmaz, akkor 4 995 000 összehasonlításra van szükség.

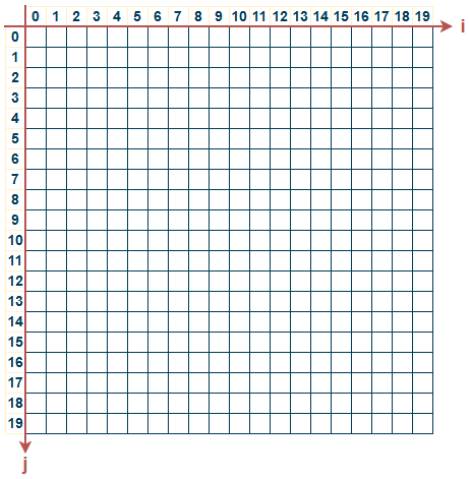
Bár a tranzitivitás létezhet bizonyos adathalmazok esetében, így például egyszerű szövegek összehasonlításánál, más esetekben nem lehet alapozni a halmaz ezen tulajdonságára. Példák ilyen esetekre:

- Ha például az adathalmaz elemei az országok, az R reláció azok szomszédságát írja le. Ilyen esetben a tranzitivitás nem fog teljesülni, tekintve, hogy ha A ország határos B országgal, és B ország határos C országgal, abból nem következik, hogy A ország is határos C országgal.

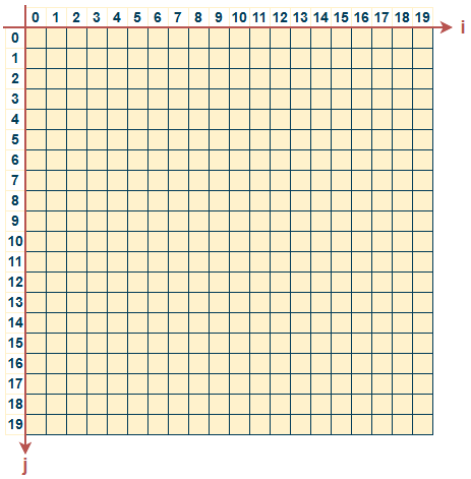
- Másik példaként legyen az adathalmaz az egydimenziós pontok halmaza, ahol az R reláció az elemek közötti távolságot írja le. Itt is igaz, hogy ha γ távolság van az A és a B pont, illetve B és C pont között, akkor nem lehet γ távolság az A és a C pont között. Míg az első példában lehetnek olyan elemei a halmaznak, ahol teljesülhet a tranzitivitás, addig ezesetben a tranzitivitás hiánya minden elemre

fennáll.

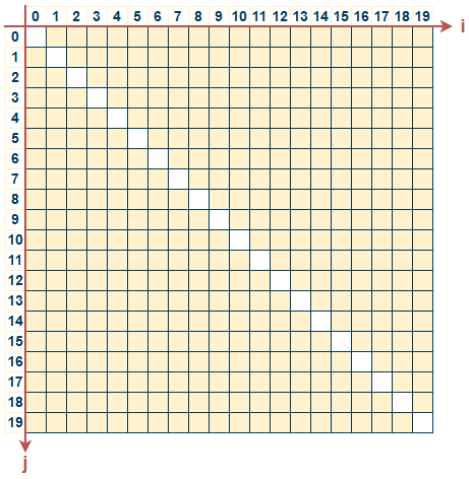
- Harmadik példaként említhető, ha az adathalmaz egy gráf csomópontjait, az R reláció a köztük lévő éleket írja le. Itt sem garantálható a tranzitivitás, így például, ha a C csúcs elérhető A-ból a B csúcson keresztül, abból nem következik, hogy az A és B csúcs között is van él: $A \longrightarrow B \longrightarrow C$.



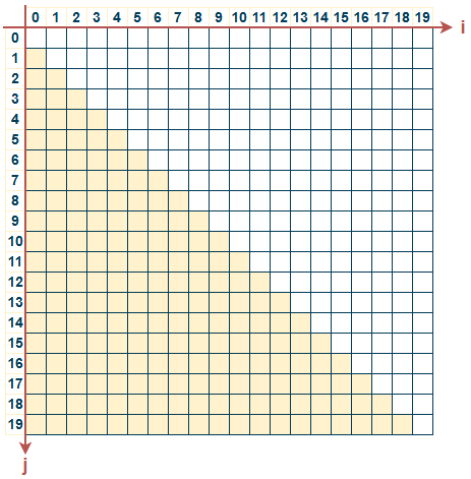
*(**A**) Az elempárok teljes mátrixa.*



*(**B**) Az elempárok mindegyikének összehasonlítása, amikor az összehasonlítások száma 400.*



*(**C**) A reflexív tulajdonság kihasználását követő összehasonlítások, melyek száma 380.*



*(**D**) A szimmetrikus tulajdonságot is fegyelembe vevő összehasonlítások, melyek száma 190.*

2.3.1. ábra. *Egy 20 elemű halmaz összehasonlítási mátrixa.*

- Végezetül utolsó példaként említhető a **2.3.2.** ábrán látható forráskódok működésük tekintetében. A **2.3.2.** (**A**) ábrán szereplő forráskód hasonlít a **2.3.2.** (**B**) ábrán szereplő forráskódra, ugyanis mindkettő külön változóban található értéket ad össze, de máshol jeleníti meg. A **2.3.2.** (**B**) ábra forráskódja hasonlít a **2.3.2.** (**C**) ábra forráskódjára, a megjelenítés, illetve a változók tekintetében (habár a változók más szerepkörben szerepelnek). A **2.3.2.** (**A**) ábra viszont több szempontból sem hasonlít a **2.3.2.** (**C**) ábrán látható forráskódra, mivel sem a változók nem tekinthetők hasonlónak, sem a kiírás.

```
int a = 5, b = 10;
int sum = a + b;
Console.WriteLine("Az összeg: " + sum);
```

*(**A**) Az 'a' és a 'b' változó összegzése és eltárolása a 'sum' változóba, majd megjelenítése a konzolon.*

```
int x = 5, y = 10;
int result = x + y;
Log.Info("Eredmény rögzítve: " + result);
```

*(**B**) Az 'x' és az 'y' változó összegzése és eltárolása a 'result' változóba, majd info szintű loggolása.*

```
string msg = "Eredmény rögzítve: ";
int prod = 5 * 10;
Log.Info(msg + prod);
```

*(**C**) A szorzat kiszámítása és eltárolása a 'prod' változóba, majd az eredmény összefűzése az 'msg' változóval és info szintű loggolása..*

***2.3.2. ábra.** Három működésükben különböző forráskód a tranzitivitás hiányának bemutatására.*

A példák alapján is látható, hogy a tranzitivitás nem használható ki, így tovább nem egyszerűsíthető a feldolgozás, ezért ellenőrizni kell a feldolgozandó halmaz összes lehetséges párját az kompatibilitási osztályok meghatározásához.

2.3.2 A szakirodalomban elterjedt módszerek

2.3.2.1 Szekvenciális megvalósítás

Egy *S* adathalmaz feldolgozása egymásba ágyazott ciklusokkal valósítható meg, ahol a ciklusok a teljes adathalmazt kétszer járják be a nyers erő módszerét alkalmazva, amelynek pszeudókódját a **2.3.1.** algoritmus mutatja be, mint *Naiv soros megvalósítás*. Az algoritmus legfőbb problémája a

lépésszám. Például, ha a halmaz mérete $N = 20$, akkor az iterációk száma (a ciklusok futóváltozóinak növelése) $F_N = N^2 = 400$, az összehasonlítások száma pedig $C_N = N^2 = 400$.

2.3.1. algoritmus. *Naiv soros megvalósítás.*

Bemenet: S – adathalmaz, N – elemek száma az adathalmazban
Kimenet: S – a feldolgozott adathalmaz

```
1:  function NaivSorosMegvalósítás(S, N)
2:      for i ← 1 to N do
3:          for j ← 1 to N do
4:              if S[i].T1 R S[j].T1 then    ▷ R a reláció két elem között
5:                  S[i].T2 =S[i].T2 ∪ {S[j]}
6:              end if
7:          end for
8:      end for
9:      return S
10: end function
```

A pszeudókódban az adathalmaz objektumokat tartalmaz, melyek rendelkeznek T_1 és T_2 tulajdonságokkal. A T_1 tulajdonság alapján kerülnek összehasonlításra az objektumok és a T_2 tulajdonságokba kerülnek összekapcsolásra a hasonló elemei a halmaznak az objektum referencia által.

Az iteráció és összehasonlítások száma jelentősen csökkenthető, ha az elemek megfelelnek a reflexivitási és szimmetrikus követelményeknek. Ezesetben elég a belső ciklust a szimmetria miatt az $i + 1$-edik elemből indítani. A **2.3.1.** algoritmus módosított változatát a **2.3.2.** algoritmus szemlélteti, mint *Javított soros megvalósítás*. A módosítás fő előnye, hogy a szimmetria miatt mindkét objektumot ugyanabban az iterációban lehet összekapcsolni, ezzel jelezve az azonosságukat. A változás miatt az iterációk száma $F_N = \frac{N*(N+1)}{2} = 210$-re, az összehasonlítások száma pedig $C_N = \frac{N^2-N}{2} = 190$-re csökken ugyanannyi $N = 20$ elem esetén.

2.3.2. algoritmus. *Javított soros megvalósítás.*

```
Bemenet:  S – adathalmaz, N – elemek száma az adathalmazban
Kimenet:  S – a feldolgozott adathalmaz
 1:  function JavítottSorosMegvalósítás(S, N)
 2:      for i ← 1 to N do
 3:          for j ← i to N do            ▷ Szimmetria tulajdonság miatt
 4:              if i ≠ j and S[i].T1 R S[j].T1 then   ▷ R a reláció
 5:                  S[i].T2 =S[i].T2 ∪ {S[j]}
 6:                  S[j].T2 =S[j].T2 ∪ {S[i]}
 7:              end if
 8:          end for
 9:      end for
10:      return S
11: end function
```

A pszeudókódban az adathalmaz objektumokat tartalmaz, melyek rendelkeznek T_1 és T_2 tulajdonságokkal. A T_1 tulajdonság alapján kerülnek összehasonlításra az objektumok és a T_2 tulajdonságokba kerülnek összekapcsolásra a hasonló elemei a halmaznak az objektum referencia által.

2.3.2.2 Egymásba ágyazott ciklusok párhuzamosítási lehetőségei

Egymásba ágyazott ciklusokat számos területen alkalmaznak (numerikus számítások, big data, stb.) matematikai képletek kiszámításához vagy nagy mennyiségű adat feldolgozásához, amelyek jellemzően (gyakran többdimenziós) tömbökben tárolódnak. Tömbök esetén az adatokhoz beágyazott ciklusokon keresztül férünk hozzá, amelyek függősége az adattípustól és a feldolgozási módszertől függ. Ilyen esetekben párhuzamosításkor először meg kell határozni, hogy a ciklusok melyik szintjén szeretnénk párhuzamosítani, amelynek stratégiáit a **2.3.1** táblázat szemlélteti. [33, 34, 35, 36]

2.3.1. táblázat. *A leggyakoribb párhuzamosítási lehetőségei az egymásba ágyazott ciklusoknak.*

Típus	Leírás
Outermost	Párhuzamosítás a legkülső ciklus szerint.
Inner	Az egyik belső ciklus párhuzamosítása.
Nested	Több egymásba ágyazott ciklus párhuzamosítása.
Collapsing	Az egymásba ágyazott ciklus átalakítása egyetlen ciklussá, majd annak a ciklusnak a párhuzamosítása.

Az ***Outermost*** párhuzamosítási megközelítés a leggyakoribb egymásba ágyazott ciklusok esetén. Ebben a stratégiában a legkülső ciklus

párhuzamosításra kerül, az iterációkat elosztva a szálak között, ezáltal a szálak párhuzamosan futnak és elvégzik a hozzájuk rendelt feladatokat. Ebben az esetben a belső ciklusok szekvenciálisan kerülnek végrehajtásra. Ez a stratégia általában jó választás (különösen nagy iterációszám esetén), mivel minimalizálja a párhuzamosítás költségeit (például a szálak inicializálása, a szálakon végrehajtott ciklusiterációk ütemezése, a szálak szinkronizálása). Az előnyök ellenére a hátránya a párhuzamosítás maximális szintjének korlátozása, amely nem haladhatja meg a párhuzamosítandó ciklus iterációinak számát. Például egy $i: 1 \rightarrow N$-ig futó ciklus maximum N részre osztható. Ez korlátozhatja a szálak számát, ami megakadályozhatja a rendszer összes processzorának és magjának kihasználását.

Az ***Inner*** alapú párhuzamosítás az előbbi módszer egy változata, azzal a különbséggel, hogy a belső ciklusok egyikét párhuzamosítják (szemben a külsővel), míg a legkülső ciklust szekvenciálisan hajtják végre. Ez a stratégia akkor szükséges vagy hasznos, ha a külső ciklus nem rendelkezik elegendő számú iterációval a hatékony párhuzamosításhoz, de hátránya, hogy a szálak inicializálása, kezelése és szinkronizálása teljesítményproblémákhoz vezethet.

A ***Nested*** párhuzamosítási stratégia kihasználja a több ciklus párhuzamos végrehajtásának lehetőségét. Az előzőekkel ellentétben, amelyek legfeljebb annyi szálat használhatnak, ahány iterációs ciklus van, ez a stratégia további párhuzamosítást is biztosít, ami jó eredményeket adhat nagyszámú processzorral rendelkező rendszereken.

A ***Collapsing*** módszer alapja, hogy a beágyazott ciklusokat átalakítja egyetlen ciklussá, majd az újonnan létrehozott ciklus kerül párhuzamosításra. Az átalakítás előnye, hogy a párhuzamosítás foka megnő, mivel az újonnan létrehozott ciklusnak több iterációja lesz. A módszer jobb futásidejű eredményeket produkál, mint az *Inner* és a *Nested* stratégiák, mivel csökkenti a ciklus overhead-jét (bár ez nem mindig lehetséges, mivel nem minden fordítóprogram biztosítja ezt a funkciót).

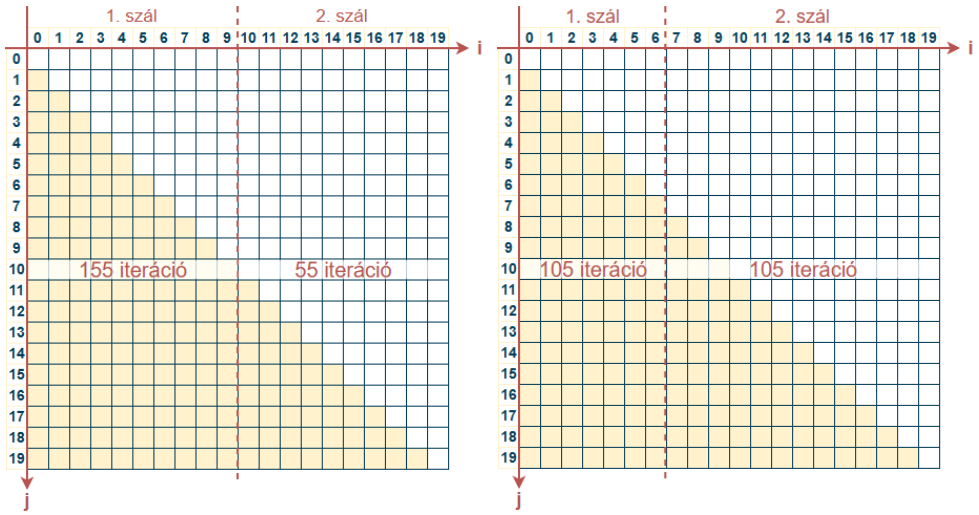
2.3.2.3 Naiv párhuzamos megvalósítás

Figyelembe véve a beágyazott ciklusokat, az *i* és a *j* futóváltozók nem hordoznak semmilyen függőséget, így jól párhuzamosíthatók. [37] Érdemes

azonban megjegyezni, hogy a *j* iterációk száma nem állandó, ez az *i* aktuális értékétől függ. Ha a párhuzamosítást a külső ciklusnak megfelelően hajtjuk végre, az eredmény a szálak kiegyensúlyozatlan futásideje lesz. Ezért más párhuzamosítási stratégiát kell választani (emennyiben meg szeretnénk tartani az eredeti megközelítést), hogy javítsuk az algoritmus futásidejét.

Amennyiben megtartanánk a külső ciklus párhuzamosítását, és a külső ciklus iterációinak száma alapján hozunk létre szálakat, akkor a probléma a szálak futási idejének különbsége lesz. Az első szál sokkal később fejeződik be, mint az utolsó, mivel a belső ciklusa a teljes adathalmazon végigiterál. Ebben az esetben a párhuzamos algoritmus teljes futási idejét az első szál határozza meg, ahogy azt a **2.3.3.** ábra és a *Naiv párhuzamosított megvalósítás* nevű pszeudokód szemlélteti a **2.3.3.** algoritmusban.

Ha egyszerűen csak szétosztanánk a számítást a külső ciklus futóváltozója szerint, akkor a szálakon történő végrehajtások száma kiegyenlítetlen lesz. Például, ha két szálra szeretnénk ez alapján bontani az $N = 20$ elemű halmaz feldolgozását, akkor az első szál 55, míg a második szál 155 iterációt hajtana végre. Ennél jobb megoldás, ha az összesített iterációkat figyelembe véve kerülne felosztásra a külső ciklus, így közel kiegyenlített futásidejű szálakat eredményezne a feldolgozás.



*(**A**) A felosztás a külső ciklus szerint történik a futóváltozó szerint egyenlő részre.*

*(**B**) A felosztás a külső ciklus szerint történik az iterációk szerint egyenlő részre.*

2.3.3. ábra. *Párhuzamosítási lehetőségei egy N=20 elemű adathalmaznak a külső ciklus szerint.*

2.3.3. algoritmus. *Naiv párhuzamosított megvalósítás.*

Bemenet: S – adathalmaz, N – elemek száma az adathalmazban
Kimenet: S – a feldolgozott adathalmaz

```
1:  function NaivPárhuzamosMegvalósítás(S, N)
2:      P = NumberOfProcessors()          ▷ Optimális szálak száma
3:      T = CreateThreads(P)              ▷ P szál létrehozása
4:      for p ← 0 to P do
5:          alsó = ⌈p*N / P⌉              ▷ alsó (inkluzív) index
6:          felső = ⌈(p+1)*N / P⌉         ▷ felső (exkluzív) index
7:          T[p]=ThreadProcess(alsó, felső, S, N)
8:      end for
9:      WaitAll(T)
10:     return S
11: end function

12: function ThreadProcess(alsó, felső, S, N)
13:     for i ← alsó to felső do
14:         for j ← i + 1 to N do
15:             if S[i].T1 R S[j].T1 then    ▷ R a reláció két elem között
16:                 S[i].T2 =S[i].T2 ∪ {S[j]}
17:                 S[j].T2 =S[j].T2 ∪ {S[i]}
18:             end if
19:         end for
20:     end for
21:     return S
22: end function
```

A pszeudókódban az adathalmaz objektumokat tartalmaz, amelyek rendelkeznek T_1 és T_2 tulajdonságokkal. A T_1 tulajdonság alapján kerülnek összehasonlításra az objektumok és a T_2 tulajdonságokba kerülnek összekapcsolásra a hasonló elemei a halmaznak az objektum referencia által.

2.3.3 Saját módszer bemutatása

2.3.3.1 Index alapú egyenlő elosztás

Az egyes szálakon belüli iterációk számának kiegyensúlyozásához meg kell határozni az optimális iterációk számát (O) egy partícióban az adathalmaz mérete (N) és a partíciók célszáma (P) alapján, és ki kell számítani az összes partíció alsó (beleértve) és felső (kizáró) indexét. Fontos megjegyezni, hogy a particionálás a külső ciklus szerint történik, így a tökéletes felbontás csak kivételes esetekben fordul elő. Minden más esetben elegendő megközelítőleg azonos számú iterációt elérni.

Egy partíció optimális iterációszámát az összes iteráció számának meghatározásával kapjuk meg (egy N elemű adathalmaz esetén): $F_N = \left\lceil \frac{N(N+1)}{2} \right\rceil$, amelyet elosztunk P-vel, hogy megkapjuk a partíció optimális iterációszámát: $O = \left\lfloor \frac{\frac{N(N+1)}{2}}{P} \right\rfloor = \left\lfloor \frac{N(N+1)}{2P} \right\rfloor$.

A korábbi példa alapján az iterációk optimális eloszlása a következőképpen határozható meg:

- $N = 20 \longrightarrow F_N = \left\lceil \frac{N(N+1)}{2} \right\rceil = \left\lceil \frac{20(20+1)}{2} \right\rceil = 210$ – az iterációk teljes száma,
- $P = 2 \longrightarrow O = \left\lfloor \frac{N(N+1)}{2P} \right\rfloor = \left\lfloor \frac{F_N}{P} \right\rfloor = \left\lfloor \frac{210}{2} \right\rfloor = 105$ – optimális iteráció 1 partíción belül.

A cél az, hogy minden szálon elérjük ezt az O iterációs számot. Tudjuk, hogy az első szál az $I_0 = 0$ (inkluzív) indexű elemek feldolgozását kezdi, az utolsó szál indexe pedig $I_2 = 20$ (exkluzív). A kérdés az, hogyan határozzuk meg a belső indexeket. Abban a példában, ahol a halmazt két szálra szeretnénk osztani, csak egy belső indexet kell definiálnunk ($I_1 = 6$ lesz), amely az első szálon inkluzív, a második szálon pedig exkluzív index lesz.

2.3.3.2 Az egyes indexek kiszámításának lépései

1. Meghatározzuk az iterációk teljes számát (F_N) az adathalmaz mérete (N) alapján: $F_N = \left\lceil \frac{N(N+1)}{2} \right\rceil$.
2. Meghatározzuk a partíciók számát (P), majd kiszámítjuk egy partíció optimális iterációszámát: $O = \left\lfloor \frac{F_N}{P} \right\rfloor$
3. Az első index ($I_0 = K_0 = 0$) ismeretében meghatározzuk a partíció következő indexének közelítését (K_1) a következő egyenletek segítségével:
    a. A lépések száma K_0-tól a halmaz végéig (K_0 ismert):

$$\frac{(N - K_0)(N - K_0 + 1)}{2} \tag{2.3.1}$$

b. Lépések száma K_1-től a halmaz végéig (K_1 ismeretlen):

$$\frac{(N-K_1)(N-K_1+1)}{2} \tag{2.3.2}$$

c. A 4.6.3 egyenletet a 4.6.1 és 4.6.2 egyenletek összevonásával és egyszerűsítésével kapjuk meg:

$$\frac{(N-K_0)(N-K_0+1)}{2} - \frac{(N-K_1)(N-K_1+1)}{2} = \frac{N(N+1)}{2}$$
$$P(N-K_0)(N-K_0+1) - P(N-K_1)(N-K_1+1) = N(N+1)$$
$$P(N-K_0)(N-K_0+1) - N(N+1) = P(N-K_1)(N-K_1+1)$$
$$\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} = (N-K_1)(N-K_1+1)$$
$$\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} = N^2 - NK_1 + N - NK_1 + K_1^2 - K_1$$
$$\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} = N^2 - 2NK_1 + N + K_1^2 - K_1$$
$$\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} = N^2 + N + K_1^2 - K_1(2N+1)$$
$$\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} - N^2 - N = K_1^2 - K_1(2N+1)$$
$$-K_1^2 + K_1(2N+1) + \frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} - N^2 - N = 0 \tag{2.3.3}$$

d. A 4.6.3 egyenlet gyökei:

$$a = -1$$
$$b = 2N+1$$
$$c = \frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} - N^2 - N \tag{2.3.4}$$

e. Ahol $b^2 - 4ac > 0$, $-\frac{b}{c}, \frac{c}{a} > 0$ és $a < 0$, vagyis [38] $X_1 \leq X_2$, ezért:

$$X_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{2.3.5}$$

f. A fentiek alapján kiszámítható a következő index, a K_1 a következők szerint:

$$K_1 = X_1 = \frac{-(2N+1) + \sqrt{(2N+1)^2 + 4\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} - N^2 - N}}{-2} \tag{2.3.6}$$

g. A valódi (exkluzív) indexeket pedig a kiszámított K_1 kerekítését követően kapjuk meg (a következő K_x index számítása a kerekítés nélküli K_{x-1} értékkel történik):

$$I_1 = \lfloor K_1 \rfloor \tag{2.3.7}$$

4. Az indexeket P alkalommal számítjuk ki, és végül így kapjuk meg a $K_P = N$ értéket.

Ez a számítás jelenti a „*Javított párhuzamos megvalósítást*”, amit a **2.3.4.** algoritmus szemléltet.

2.3.3.3 Példa a közbenső indexek kiszámítására

A példában az adathalmaz mérete legyen $N = 8$, amelyet $P = 4$ partícióra szeretnénk bontani. Ezesetben az iterációk száma $F_N = \left\lceil \frac{N(N+1)}{2} \right\rceil = \left\lceil \frac{8(8+1)}{2} \right\rceil = 36$. Egy partíció optimális iterációjának száma $O = \left\lfloor \frac{F_N}{P} \right\rfloor = \left\lfloor \frac{36}{4} \right\rfloor = 9$. A **2.3.4**-es egyenlet alapján az $a = -1$ konstans, a $b = 2N + 1 = 2 * 8 + 1 = 17$, ami az N-től függ és a számítás kezdetén meghatározható. Az egyetlen érték a c, amelyet minden iterációban meg kell határozni. A külső ciklus futóváltozójának kezdő értéke $I_0 = K_0 = 0$. A közbenső indexek számítása a következő:

- 1. lépés – a K_1 és az I_1 kiszámítása:

$$c_1 = \frac{4 * 8 * 9 - 8 * 9}{4} - 8^2 - 8 = -18$$
$$K_1 = \frac{-17 + \sqrt{17^2 - 4 * -1 * -18}}{-2} = 1{,}135$$
$$I_1 = 1 \tag{2.3.8}$$

  Az első szál esetében a külső ciklus 0-tól 1-ig fog iterálni, ami összesen 8 iterációt eredményez.

- 2. lépés – a K_2 és az I_2 kiszámítása:

$$c_2 = -36 \quad K_2 = 2{,}479 \quad I_2 = 2 \tag{2.3.9}$$

  A második szál esetében a külső ciklus 1-től 2-ig iterál, ami összesen 7 iterációt eredményez.

- 3. lépés – a K_3 és az I_3 kiszámítása:

$$c_3 = -54 \quad K_3 = 4{,}227 \quad I_3 = 4 \tag{2.3.10}$$

  A harmadik szálnál a külső ciklus 2-től 4-ig iterál, ami összesen 11 iterációt eredményez.

- 4. lépés – a K_4 és az I_4 kiszámítása:

$$c_4 = -74 \quad K_4 = 8{,}000 \quad I_4 = 8 \tag{2.3.11}$$

A negyedik szálnál a külső ciklus 4-től 8-ig iterál, ami összesen 10 iterációt eredményez.

2.3.4. algoritmus. *Javított párhuzamosított megvalósítás.*

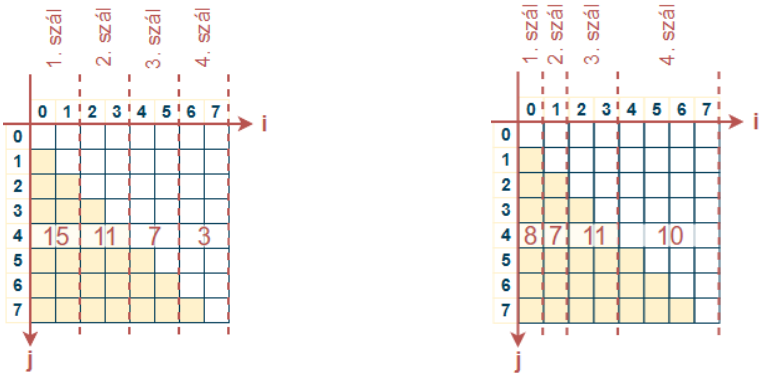
```
Bemenet:  S – adathalmaz, N – elemek száma az adathalmazban
Kimenet:  S – a feldolgozott adathalmaz
 1: function JavítottPárhuzamosMegvalósítás(S, N)
 2:     P = NumberOfProcessors()          ▷ Optimális szálak száma
 3:     T = CreateThreads(P)              ▷ P szál létrehozása
 4:     a = -1
 5:     b = 2N + 1
 6:     K_alsó = 0
 7:     I_alsó = 0
 8:     for p ← 0 to P do
 9:         c = (P(N−K_alsó)(N−K_alsó+1)−N(N+1))/P − N^2 − N
10:         K_felső = (−b+√(b^2−4ac))/2a
11:         I_felső = ⌊K_felső⌋           ▷ felső (exkluzív) index
12:         T[p]=ThreadProcess(I_alsó, I_felső, S, N)
13:         K_alsó = K_felső
14:         I_alsó = I_felső
15:     end for
16:     WaitAll(T)
17:     return S
18: end function

19: function ThreadProcess(alsó, felső, S, N)
20:     for i ← alsó to felső do
21:         for j ← i + 1 to N do
22:             if S[i].T1 R S[j].T1 then   ▷ R a reláció két elem között
23:                 S[i].T2 =S[i].T2 ∪ {S[j]}
24:                 S[j].T2 =S[j].T2 ∪ {S[i]}
25:             end if
26:         end for
27:     end for
28:     return S
29: end function
```

A pszeudókódban az adathalmaz objektumokat tartalmaz, melyek rendelkeznek T_1 és T_2 tulajdonságokkal. A T_1 tulajdonság alapján kerülnek összehasonlításra az objektumok és a T_2 tulajdonságokba kerülnek összekapcsolásra a hasonló elemei a halmaznak az objektum referencia által.

Mint látható, nem érhetjük el az egyenletes eloszlást, de közel kerülünk hozzá. Az egy szálra vonatkozó optimális iterációs szám $O = 9$. Az eltérések ettől az optimális értéktől a következők: -1 (-11%), -2 (-22%), $+2$ (22%), $+1$ (11%), 0 kumulatív hibával.

Ha ugyanezt a problémát szeretnénk a „*Naiv párhuzamosítással*” megoldani akkor a szálak rendre a következő indexekkel dolgoznának: $0-2$, $2-4$, $4-6$, $6-8$, ahol az iterációk számai a következők lennének: 15 (44%), 11 (22%), 7 (-22%), 3 (-44%), szintén 0 kumulatív hibával. Jól látható, hogy az első szál körülbelül ötször tovább fog futni, mint az utolsó. Ezzel szemben az általam kifejlesztett modell szálai körülbelül ugyanabban az időben fognak befejeződni, ahogy az a **2.3.4.** ábrán is látható.



*(**A**) Naiv párhuzamos megvalósítás esetén az egyes szálakra jutó iterációk száma.* *(**B**) Javított párhuzamos megvalósítás esetén az egyes szálakra jutó iterációk száma.*

2.3.4. ábra. *Az egyes szálakra jutó iterációk különbsége a két párhuzamosítási lehetőség között.*

2.3.3.4 Hiba az indexek kiszámításában

Vannak esetek, amikor a számítási pontatlanság miatt soha nem érhető el az utolsó index. Mivel az alsó egészre kerekítés fontos lépés az indexek definiálásában, ez a probléma akkor jelentkezik, amikor a hiba hatására a következő egész fölé kerülünk, vagyis egyel nagyobb értéket kapunk az ideálisnál, így ezután a kerekítési művelet nem lesz képes korrigálni az eltérést, legalább eggyel nagyobb értékre fog kerekíteni az elvártnál. Ez a hiba csak az utolsó index kiszámításakor jelentkezik ténylegesen, mivel a közbenső indexek esetében ez a különbség elhanyagolható, minimális eltéréseket okoz az iterációkban.

Példaként tegyük fel, hogy a halmaz mérete $N = 350\,000\,000$, a partíció pedig $P = 8$, ahol az optimális iteráció egy szálon belül $O = 7\,656\,250\,021\,875\,000$. Ebben az esetben a számítás után az indexek a **2.3.2.** táblázatban láthatók.

Az utolsó számításnál az index a szükséges érték alatt van, ezért korrigálni kell. Ennek kiküszöbölésére megoldást jelenthet, ha az indexeket csak a $P - 1$-edik partícióig számítjuk ki, majd az utolsó partíciónál az utolsó indexet N-nek választjuk.

2.3.2. táblázat. *Indexek eloszlása N= 350 000 000 és P=8 esetén.*

Index alsó értéke	Index felső értéke	Iterációk száma	Százalékos eltérés
0	22 604 979	7 656 250 123 507 270	0,0000013274%
22 604 979	46 891 109	7 656 249 998 313 340	-0,0000003077%
46 891 109	73 300 705	7 656 249 988 081 220	-0,0000004414%
73 300 705	102 512 627	7 656 250 044 133 910	0,0000002907%
102 512 627	135 669 648	7 656 250 019 577 120	-0,0000000300%
135 669 648	175 000 000	7 656 249 913 887 130	-0,0000014105%
175 000 000	226 256 314	7 656 250 113 194 860	0,0000011927%
226 256 314	349 999 995	7 656 249 974 305 130	-0,0000006213%

2.3.3.5 A partíció maximális hasznos mérete

A maximális hasznos partíció azt a P értéket jelöli, ahol minden partícióhoz legalább egy külső ciklus iterációja végrehajtásra kerül (az $I_{also} < I_{felső}$ feltétel teljesül). Ez a feltétel megakadályozza, hogy olyan szálakat hozzunk létre, amelyek nem tartalmaznak iterációkat. Ez a feltétel általában akkor sérül, ha a feldolgozni kívánt halmaz túl kicsi, vagy ha a párhuzamosítás szintje túl magas.

Vegyük például az előző példát, ahol $N = 8$ és $P = 6$. Ebben az esetben a kiszámított indexek a következők lesznek: $0 - 1$, $1 - 2$, $\mathbf{2 - 2}$, $2 - 4$, $4 - 5$, $5 - 8$, ahol a harmadik szál indexei megsértik a feltételt: mivel I_{also} és $I_{felső}$ indexeinek értéke 2, ez a szál semmit sem fog tenni.

A probléma kiküszöbölésére érdemes a P értékét egy $alsó$ és egy $felső$ határérték közé szűkíteni. Az $alsó$ határértéket az $\frac{N+1}{2}$ képlettel kapjuk meg, mivel az elemek számának legalább felére partíciókat tudunk létrehozni. Ez az egyenlet a következő összefüggésből származtatható:

$$N \geq \frac{N(N+1)}{2P}$$
$$1 \geq \frac{N+1}{2P}$$
$$P \geq \frac{N+1}{2} \tag{2.3.12}$$

A felső határértéket az alsó határérték kiterjesztése után a következő képlettel határozhatjuk meg:

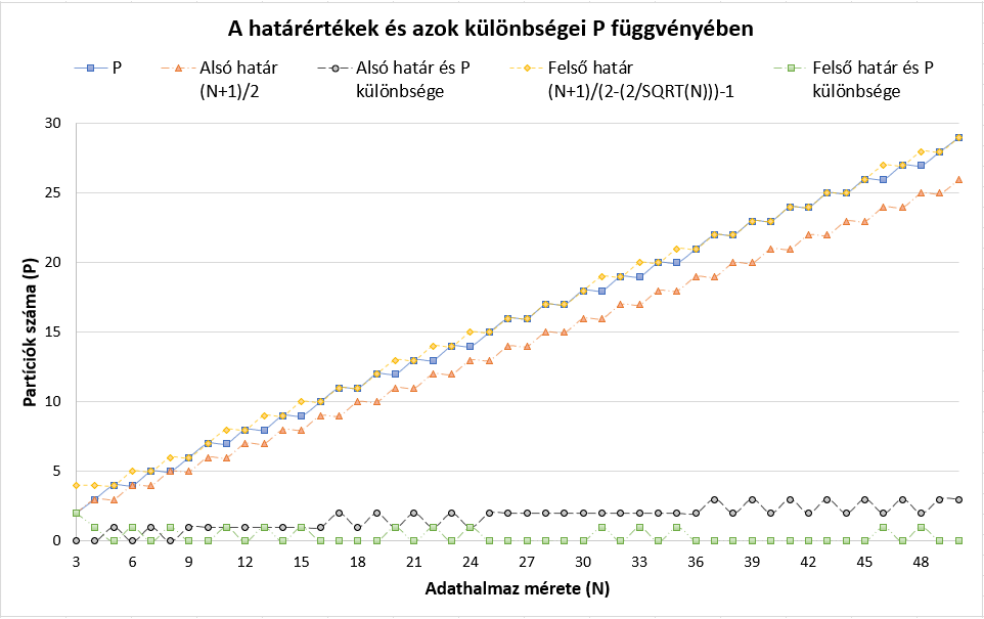
$$P < \frac{N+1}{2 - \frac{2}{\sqrt{N}}} - 1 \tag{2.3.13}$$

A **2.3.12** és **2.3.13** határérték egyenletek alapján az első 50 partícióérték a **2.3.3.** táblázatban és a **2.3.5.** ábrán láthatók. Amint látható, a $felső$ határ a valódi maximális P értéket követi (néhány speciális esettől eltekintve), az $alsó$ határ pedig lassan távolodik tőle.

Összefoglalva, ha a P értéket az $alsó$ határértéknél kisebbre vagy azzal egyenlőre választjuk, a particionálás minden bizonnyal helyes lesz, és minden szál feldolgozható iterációkat fog tartalmazni. Magasabb értékek esetén a $alsó$ határérték és az optimális érték közötti különbség növekszik, ezért célszerű a $felső - 1$ határértéket használni a magas szintű párhuzamosság elérése érdekében.

2.3.3. táblázat. *A határértékek és azok különbségeinek alakulása a P függvényében.*

N	P	Alsó határ		Felső határ		N	P	Alsó határ		Felső határ	
		Érték	Eltérés	Érték	Eltérés			Érték	Eltérés	Érték	Eltérés
3	2	2	*0*	4	*2*	27	16	14	*2*	16	*0*
4	3	3	*0*	4	*1*	28	17	15	*2*	17	*0*
5	4	3	*1*	4	*0*	29	17	15	*2*	17	*0*
6	4	4	*0*	5	*1*	30	18	16	*2*	18	*0*
7	5	4	*1*	5	*0*	31	18	16	*2*	19	*1*
8	5	5	*0*	6	*1*	32	19	17	*2*	19	*0*
9	6	5	*1*	6	*0*	33	19	17	*2*	20	*1*
10	7	6	*1*	7	*0*	34	20	18	*2*	20	*0*
11	7	6	*1*	8	*1*	35	20	18	*2*	21	*1*
12	8	7	*1*	8	*0*	36	21	19	*2*	21	*0*
13	8	7	*1*	9	*1*	37	22	19	*3*	22	*0*
14	9	8	*1*	9	*0*	38	22	20	*2*	22	*0*
15	9	8	*1*	10	*1*	39	23	20	*3*	23	*0*
16	10	9	*1*	10	*0*	40	23	21	*2*	23	*0*
17	11	9	*2*	11	*0*	41	24	21	*3*	24	*0*
18	11	10	*1*	11	*0*	42	24	22	*2*	24	*0*
19	12	10	*2*	12	*0*	43	25	22	*3*	25	*0*
20	12	11	*1*	13	*1*	44	25	23	*2*	25	*0*
21	13	11	*2*	13	*0*	45	26	23	*3*	26	*0*
22	13	12	*1*	14	*1*	46	26	24	*2*	27	*1*
23	14	12	*2*	14	*0*	47	27	24	*3*	27	*0*
24	14	13	*1*	15	*1*	48	27	25	*2*	28	*1*
25	15	13	*2*	15	*0*	49	28	25	*3*	28	*0*
26	16	14	*2*	16	*0*	50	29	26	*3*	29	*0*



2.3.5. ábra. *Az alsó határ és a felső határ alakulása az adathalmaz méretének függvényében, összehasonlítva a P partíciószámával, melyek közötti eltérések külön is ábrázolva vannak. Az ábra jól szemlélteti, hogy a felső határ szorosan követi a P értéket még nagyobb adathalmaz méret esetén is, míg az alsó határ és a P érték közötti eltérés minimálisan emelkedik az adathalmaz méretének növekedésével.*

2.3.3.6 A bemutatott módszer értékelése

A bemutatott algoritmusok futásideje a következő paraméterek szerint generált bemenetekkel kerültek vizsgálatra (mely összesen 96 darab adathalmazt eredményezett):

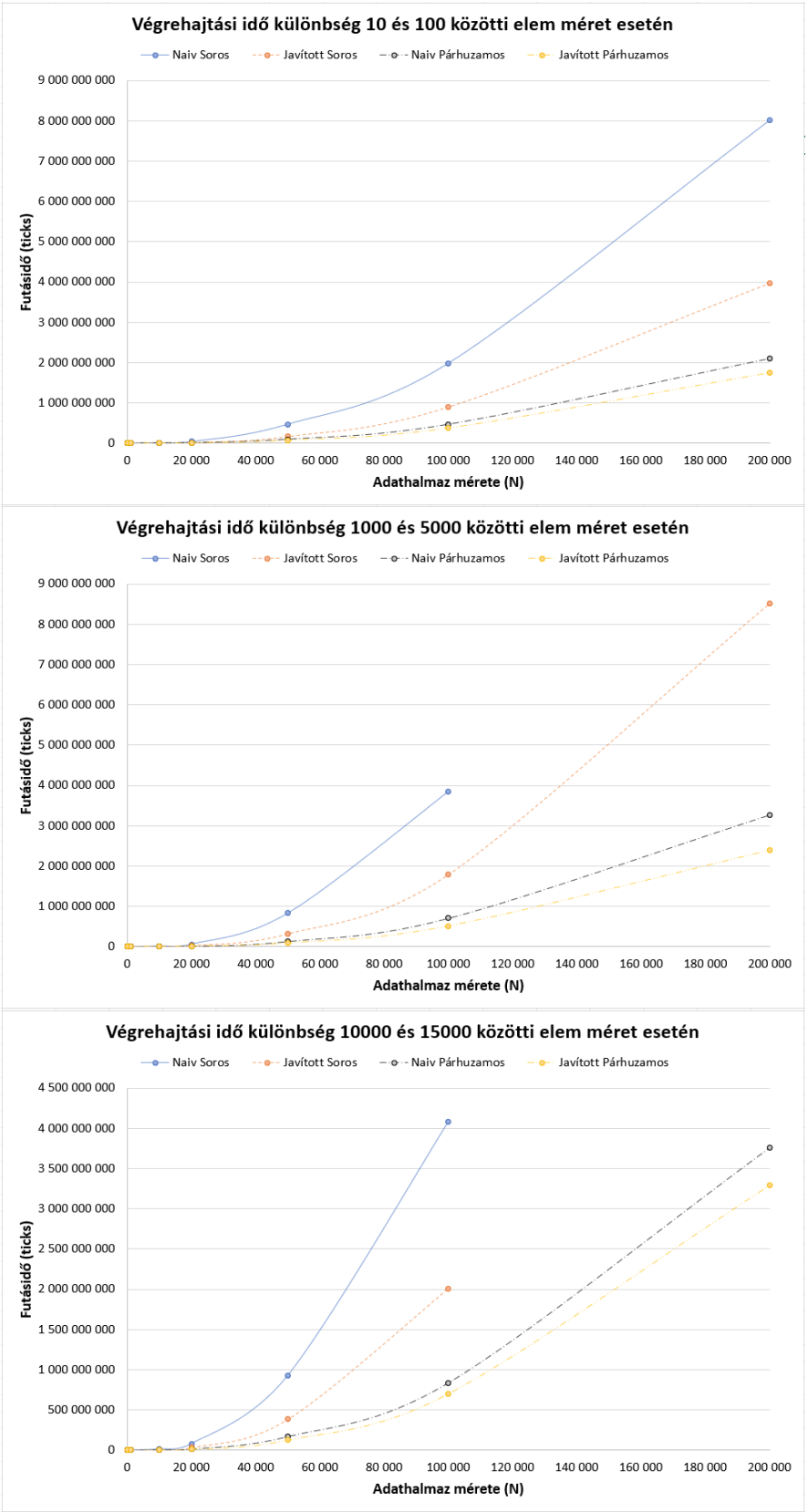
- Az adathalmaz mérete (N) értékei: 10, 100, 1 000, 10 000, 20 000, 50 000, 100 000, 200 000
- Az adathalmaz elemeinek minimális és maximális „méretei" (E_l): 10-100, 1 000-5 000, 10 000-15 000
- Az ismétlődések valószínűsége az adathalmazban (β): 0, 0.33, 0.5, 0.8

A teszteléshez egy C# alkalmazás került kifejlesztésre, amely minden mérést 10-szer futtat le minden algoritmussal. A mérések során a legjobb és a legrosszabb eredményeket figyelmen kívül hagytam, a végeredmény a fennmaradó értékek átlaga lett.

A teszteléshez használt konfiguráció:

- CPU: Intel(R) Core(TM) i5-7300 (2 physical cores, 4 logical cores)
- RAM: 16 GB
- HDD: Samsung MZVLB512HAJQ-000L7, 475,48 GB
- FileSystem: NTFS v3.1, cluster size: 4096 bytes
- Used level of parallelizm (P): 4

A mérési eredmények azt mutatják, hogy a halmazon belüli elemek duplikációjának mértéke (β) és az elemek „mérete" (E_l) nem befolyásolja jelentősen a futásidőt. A bemutatott négy algoritmus futásidő különbségét az adathalmazok méretének különbségei határozzák meg. Ezt a megfigyelést illusztrálják a mérések eredményei a **2.3.6.** ábrán.



2.3.6. ábra. *A naiv és javított, soros és párhuzamos algoritmusok futásidejének*

összehasonlítása kis (10–100), közepes (1 000–5 000) és nagy (10 000–15 000) elem méret esetén. A javított párhuzamos implementáció már csekély elem méret esetén is kedvezőbb végrehajtási időt mutatott a többi algoritmushoz képest, az elemek és az adathalmaz méretének növelésével pedig ez az előny progresszív módon tovább fokozódott.

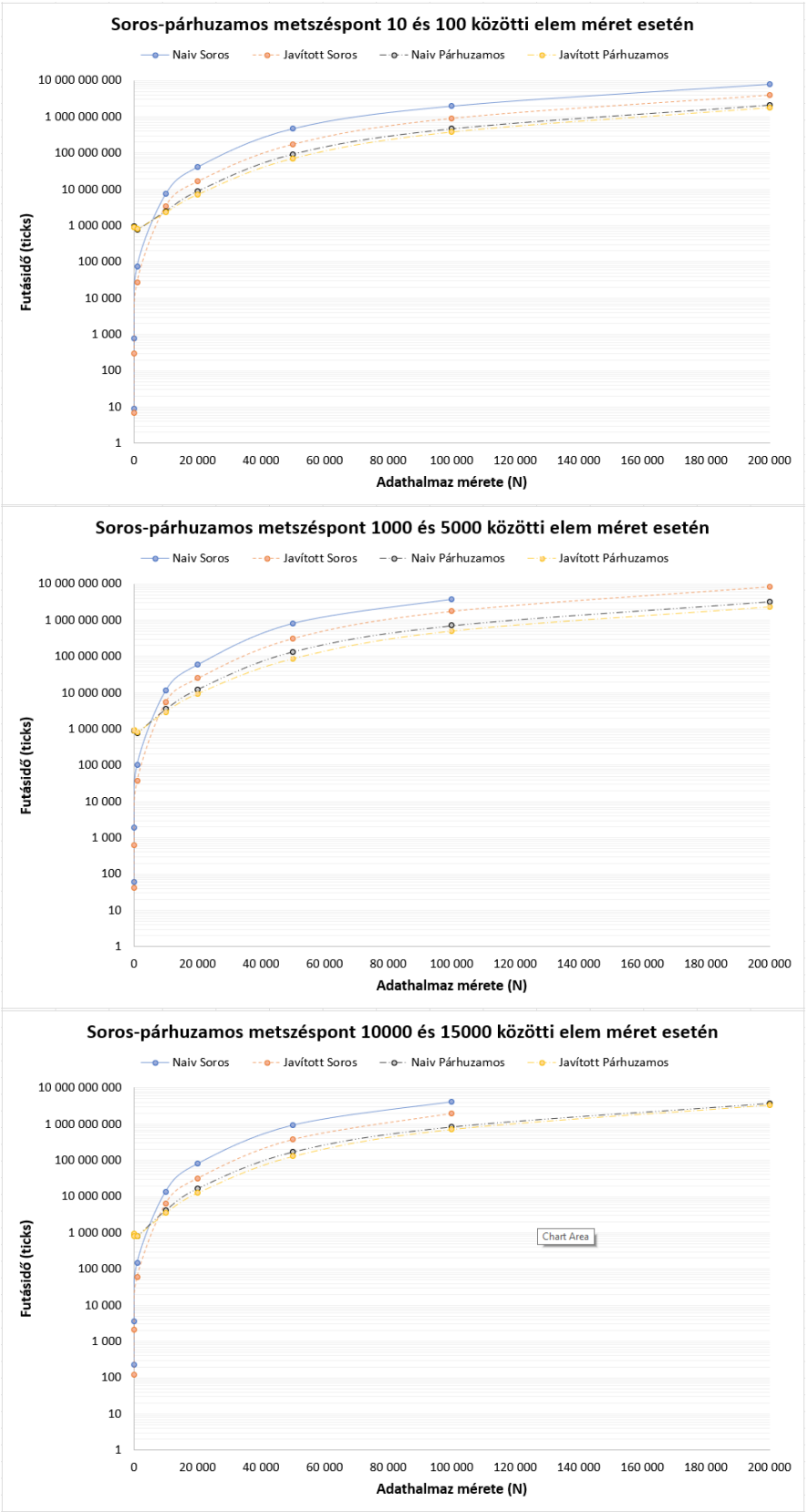
Az ábrákon látható, hogy a függvények meredeksége közel azonos. Például a második ($E_l: 1000 - 5000$) és a harmadik ($E_l: 10000 - 15000$) diagramban a Naiv Soros algoritmus végrehajtási idejét összevetve látható, hogy bár a második esetben a halmaz elemei kétszer akkorák, a futásidő a halmaz elemszáma miatt növekedett. Például 50 000 elemnél az elemek méretének növelésekor a különbség a kettő eset (második és harmadik) futásidejének különbsége csak 100 927 317 tick volt (~kb 0,010092732 másodperc), addig a halmaz méretének megduplázásakor (második diagrammon, 50 000 elem helyett 100 000 elem esetén) a végrehajtási idő átlagosan 3 012 800 705 tickkel nőtt (~0,301280079 másodperc). Természetesen ez az összefüggés más mérésekre is vonatkozik.

A mérések alapján az is meghatározható (ahogy a **2.3.7.** ábrán látható), hogy több mint 10 000 elem esetén érdemes a párhuzamosítást alkalmazni.

Azokban az esetekben, amikor az adatkészlet mérete nem éri el ezt az elemszámot ($N \geq 10\,000$), a soros feldolgozás kisebb végrehajtási időt eredményez, függetlenül az elemek hosszától.

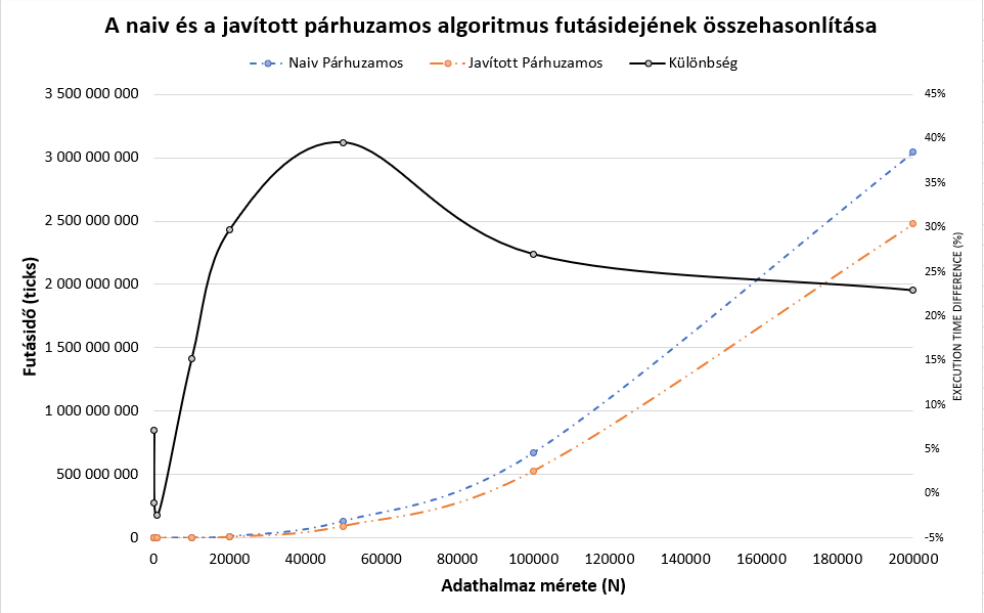
Végezetül a **2.3.8.** ábrán látható a párhuzamos algoritmusok két változatának összehasonlítása, kiemelve a futási idők közötti különbségeket.

Mint látható, azokban az esetekben, amikor az adathalmaz mérete meghaladja a párhuzamosításhoz szükséges minimális elemszámot ($N \geq 10\,000$), a párhuzamos algoritmus javított változata mindig kisebb végrehajtási időt eredményez (15 - 40 % között), mint a naiv verzió.



2.3.7. ábra. *A soros és párhuzamos algoritmusok futásidejének alakulása az*

adathalmaz méretének függvényében, három különböző elemméretet (10–100, 1 000–5 000 és 10 000–15 000) vizsgálva. Az eredmények azt mutatják, hogy míg kis adathalmazoknál (N<10 000) a soros implementációk teljesítménye jobb, az adathalmaz méretének növelésével a párhuzamos algoritmusok – különösen a javított változat – szignifikánsan alacsonyabb futásidőt eredményez.



2.3.8. ábra. *A naiv és a javított párhuzamos algoritmusok végrehajtási idejének összehasonlítása az adathalmaz méretének függvényében, valamint az ezek közötti százalékos különbségek. A javított algoritmus a legtöbb tartományban hatékonyabb, a végrehajtási idő különbsége N=50,000 körül éri el a maximumát (közel 40%), majd enyhén mérséklődik a nagyobb adathalmazok esetén.*

2.3.4 Konklúzió

Az újonnan bemutatott egymásba ágyazott ciklusok index alapú párhuzamosításának módszere alkalmas egy *R* reláció kiszámításának felgyorsítására. A módszerrel a particionálás a halmaz elemeinek száma, az optimális iteráció, és a párhuzamosítás szintjének figyelembevételével történik.

Az ideális particionálás érdekében meghatározásra került egy alsó és egy felső határérték, melyet a tétlen szálak elkerülése érdekében célszerű figyelembe venni. Mindezek mellett méréseket végeztem annak megállapítására, hogy mekkora adathalmaz és azon belül milyen tulajdonságok alapján érdemes használni ezt a módszert.

2.3.5 Tézis kimondása

1.3. tézis – Kidolgoztam egy újszerű particionálási eljárást, ami alkalmas egy halmaz elemeinek reflexív, szimmetrikus, nem tranzitív reláció alapú kompatibilitási osztályokba sorolásának hatékony párhuzamosítására. Az általam kidolgozott módszer a szükséges összehasonlító műveleteket a létező hasonló megoldásokhoz képest jobb terheléselosztással valósítja meg. A módszer igazolásához számos tesztet futtattam, amelyek igazolták, hogy egy minimális elemszám felett az általam kidolgozott particionálás jelentősen kisebb futásidőhöz vezet, mint a meglévő hasonló felosztó módszerek. [S3]

2.4 Eredmények értékelése

A **2.1.** fejezetben bemutatott módszer, mely a forráskódok közötti hasonlóság mérésére szolgál, a leghosszabb közös részstring (Longest Common Substring, LCS) elvén alapul. Ez a módszer a korábbi, karakterlánc- és szószintű metrikáknál hatékonyabban képes figyelembe venni a forráskódok sajátosságait. A vizsgált módszer nagy előnye, hogy a forráskódok esetében gyakran előforduló utasítások sorrendjének felcserélhetőségét nem tekinti jelentős különbségnek, így képes helyesen kezelni azokat az eseteket, amikor két program funkcionálisan azonos, de eltérő sorrendű utasításokat tartalmaz, vagyis felépítése különböző. Ezzel szemben a hagyományos szövegalapú metrikák, mint a Koszinusz hasonlóság, Jaccard-index vagy Levenshtein-távolság, az ilyen típusú változtatásokat jelentős eltérésként kezelik, és ezért pontatlan eredményt adnak.

Az LCS-alapú megközelítés emellett kellően egyszerű implementációt tesz lehetővé, programozási nyelv független módon. Nem igényel nyelvspecifikus előfeldolgozást vagy szintaktikai elemzést, így széles körű alkalmazhatóságot biztosít. Az eredmények szerint a hasonlósági értékek pontosabban tükrözik a valós hasonlóságot, mint a többi módszer. Ugyanakkor hátrányként említhető, hogy a részstringek vágása és a többszöri futtatási igény miatt a módszer időigényes lehet nagyobb bemeneti adathalmaz esetén. Emellett előfordulhat, hogy egyes speciális esetekben a módszer túl magas hasonlósági értéket ad, például, ha egy teljes kódrészlet beágyazódik (megtalálható) egy másik kódba, anélkül, hogy valódi szerkezeti azonosság lenne.

A **2.2.** fejezetben részletezett LCS-algoritmus GPU-alapú adatpárhuzamosítása jelentős előrelépést jelent a feldolgozási idő csökkentése terén. A GPU-s megvalósítás lehetővé teszi a feladatok párhuzamos végrehajtását, különösen hosszú bemeneti karakterláncok esetében. A mérési eredmények szerint 130 karakternél hosszabb szövegeknél már érezhető a gyorsulás, ami igazolja a GPU-k hatékonyságát a szövegösszehasonlító feladatokban. Emellett hátrányként említhető, hogy a párhuzamosított modell bizonyos esetekben más eredményt ad a CPU-s változathoz képest a GPU kód indeterminisztikus viselkedése következtében, a mért eltérések azonban elfogadható mértékűek.

A **2.3.** fejezetben bemutatott egymásba ágyazott ciklusok indexfüggő párhuzamosítása szintén figyelemreméltó eredményeket hozott. A párhuzamos végrehajtás lehetővé tette a reflexív, szimmetrikus, nem tranzitív relációk gyors kiszámítását nagy adathalmazokon. A tesztek igazolták, hogy az új particionálási stratégia – amely az optimális alsó és felső határértékek figyelembevételével osztja fel a feladatokat a szálak között – jelentős futásidő-megtakarítást eredményezett, különösen nagyobb adathalmazok esetében. Az eredmények szerint a párhuzamosított verzió 10 000 elem felett mindig gyorsabb volt a soros változatnál, míg kisebb adathalmazoknál a párhuzamosítás költségei nem térültek meg. A futási idő csökkenése 15–40%-os sávban mozgott a tesztelt esetekben, ami lényeges teljesítménynövekedést jelentett. A módszer korlátját a szálak számának és a particionált adathalmaz méretének aránya szabja meg, így érdemes tovább vizsgálni az adaptív particionálási stratégiák alkalmazását, amelyek a bemenet jellemzői alapján döntenek a párhuzamosítási szintről.

2.4.1 Továbbfejlesztési lehetőségek, jövőbeni kutatási irányok

A bemutatott eredmények alapján több lehetséges irány is felmerült a kutatás folytatására, amelyek a módszerek hatékonyságának, pontosságának és általánosíthatóságának növelését célozzák.

2.4.1.1 Az LCS-alapú forráskód-összehasonlítás finomítása

Az LCS-alapú módszer fő hátránya, hogy a karakterláncok közötti legnagyobb közös részsorozatra koncentrál, ami egyes esetekben félrevezetően magas hasonlósági értéket eredményezhet. Ennek kiküszöbölésére érdemes lehet egy kombinált megközelítés kidolgozása, amely egyszerre veszi figyelembe a karakter- és token-szintű hasonlóságot. A tokenizálás bevezetése lehetővé tenné a program szerkezeti elemeinek figyelembevételét is (pl. változónevek, függvényhívások), de ennek ára a nyelvspecifikus előfeldolgozás szükségessége lenne.

2.4.1.2 GPU-alapú modell skálázhatóságának bővítése

A GPU-s megvalósítás esetében (annak sajátosságaiból fakadóan) a memóriaátviteli késleltetés és a kernelindítási idő csökkentése kulcsfontosságú fejlesztési irány. Célszerű lenne több szöveg egyidejű

összehasonlítását megvalósítani úgy, hogy a bemeneteket csak egyszer kelljen átmásolni a GPU memóriájába. Ez különösen akkor lenne előnyös, ha több elem páros összehasonlítását kell elvégezni, hiszen így az adatátvitel hatása jelentősen csökkenthető. Továbbá az átfedő adatmozgatás és kernelindítás módszerének alkalmazása is csökkenthetné a várakozási időket.

2.4.1.3 Párhuzamos cikluskezelés automatizálása és adaptivitása

A párhuzamos ciklusok esetében érdemes lenne egy olyan bemenet-vezérelt particionáló algoritmus kidolgozása, amely automatikusan választja meg a jelenleginél megfelelőbb párhuzamosítási szintet és stratégiát a bemeneti adathalmaz mérete, szerkezete és redundanciája alapján. A jelenlegi statikus küszöbértékek helyett így dinamikus, futás közben meghatározott értékek alapján történhetne a feldolgozás optimalizálása, javítva a szálak kihasználtságát, illetve csökkentve az azok közötti szinkronizációs költségeket.

Összegzésképpen elmondható, hogy a 2. fejezetben bemutatott módszerek jelentős előrelépést képviselnek a forráskódok hatékony összehasonlításában és a párhuzamosítási technikák alkalmazásában. A javasolt továbbfejlesztési irányokkal ezek a megoldások még szélesebb körben, nagyobb hatékonysággal és pontossággal alkalmazhatók lesznek a jövő informatikai kihívásainak megoldására.

3 Gépi tanuláson alapuló módszerek

3.1 Rövid szöveges válaszok kiértékelése neurális hálózattal

3.1.1 Bevezetés

A magyar nyelvű rövid szöveges válaszok feldolgozására az idők során számtalan megoldás született. [39] A neurális hálózatok térhódítását megelőzően a gépi tanuláson alapuló szövegosztályozási feladatokban leggyakrabban a klasszikus algoritmusok (mint például a tartóvektor-gép (SVM), a logisztikus regresszió vagy a döntési fák) játszottak központi szerepet. [40] Ezek a megközelítések jellemzően a természetes nyelvű feldolgozás (NLP) hagyományos eszköztárára támaszkodtak a szövegek numerikus reprezentációja során. Széles körben alkalmazott módszernek számít a szózsák modell (Bag-of-Words), az n-gramok használata, vagy a TF-IDF (Term Frequency-Inverse Document Frequency) [41] alapú súlyozás, amelyek a szavak előfordulási gyakoriságát, illetve az adott dokumentumra jellemző egyediségét számszerűsítik. Bár ezek a hagyományos gépi tanulási modellek felépítésüket tekintve egyszerűbbek és számítási szempontból kevésbé erőforrás-igényesek, jelentős hátrányuk, hogy a szöveg kontextusát, a szavak közötti finomabb szintaktikai és szemantikai kapcsolatokat, valamint a szórend módosító hatását nehezen vagy egyáltalán nem képesek megérteni. Ezeknél a módszereknél a bemenet kialakítása jellemzően merev, és a magas dimenziószámú, ritka mátrixok gyakran rontják az osztályozás hatékonyságát.

A konvolúciós neurális hálózatok (CNN) [42] alkalmazása ezen a területen jelentős paradigmaváltást hozott, és a jelen kutatásban történő kiválasztása éppen a klasszikus módszerek korlátainak leküzdésével indokolható. Míg a hagyományos osztályozók esetében a jellemzőkinyerés (feature extraction) egy rendkívül komplex, manuális szabályokhoz és szótárakhoz kötött előfeldolgozási lépés, a CNN-ek a bemeneti szöveget egy folyamatos vektortér elemeiként értelmezik. Ennek a megközelítésnek a legnagyobb előnye, hogy a hálózat konvolúciós szűrői, amelyek jellemzően a mátrix teljes sorát (vagyis a teljes szót) lefedik, képesek a lokális összefüggések, a kontextus és a szavak sorrendjéből adódó mögöttes jelentés automatikus

felismerésére és kinyerésére. A neurális hálózatok fejlődésével így lehetőség nyílt egy lépésben eldönteni egy szövegről, hogy az válasza-e a kérdésnek vagy sem, mindenféle bonyolult előfeldolgozás nélkül. Ez az architekturális tulajdonság teszi a CNN-t lényegesen alkalmasabbá arra, hogy a merev kulcsszókeresési vagy pusztán szógyakoriságon alapuló technikák helyett a válaszok tényleges tartalmát és szakmai összefüggéseit értékelje ki.

A jelen fejezetben vizsgált, szóbeágyazásokra (word embeddings) és konvolúciós neurális hálózatokra (CNN) épülő architektúrák jó alapot biztosítanak a természetes nyelvű válaszok automatikus kiértékeléséhez, azonban fontos kontextusba helyezni a módszert a 2018 után megjelent, paradigmaváltó kutatásokkal. A természetes nyelvfeldolgozás területe az elmúlt években a figyelemmechanizmusokra épülő Transformer architektúrák [43], majd az ezekre épülő mélyen kontextualizált, előtanított nyelvi modellek (mint a például a BERT [44] és a RoBERTa [45]) térnyerésével teljesen átalakult. Napjaink state-of-the-art megoldásait a nagyméretű, generatív nyelvi modellek (LLM-ek), például a GPT architektúrák [46, 47] uralják, amelyek zero-shot vagy few-shot megközelítéssel emberi szintű teljesítményt érnek el a szövegértésben. A legújabb modellekhez viszonyítva a bemutatott CNN-alapú módszer egy erőforrás-hatékony alternatívát képvisel. Míg az LLM-ek alkalmazása hatalmas számítási kapacitást és dedikált hardveres (GPU/TPU) infrastruktúrát igényelnek, továbbá fekete dobozként működve ki vannak téve a hallucináció kockázatának [48], addig az általam bemutatott CNN modell betanítása és felhasználása nagyságrendekkel kisebb erőforrásigényű. Továbbá habár a természetes nyelvű feladatok megoldása eltér a forráskódok vizsgálatától, a fejezetben bemutatottak jó alapját jelentik a forráskódok szövegalapú feldolgozásának.

3.1.1.1 Feladattípusok osztályozása

A feladatokat két nagy csoportba lehet sorolni a válasz típusa alapján:

- *Passzív feladatok*: A válaszok előre definiáltak, a válaszoló kiválaszthat egyet vagy többet a lehetőségek közül. A kiértékelő algoritmus a megjelölt válaszokat egy jó válaszhalmazzal hasonlítja össze.

- *Aktív feladatok*: A válasz szabad megfogalmazású, így csak a szintaxisa jósolható meg. Ilyen például egy matematikai kifejezés. Ez a csoport további négy alfeladat típusra bontható:

    - *Kihagyásos feladat*: A kihagyásos feladatokat jellemzően nyelvtani tesztekben használják, ahol a válasz szó(/szavak), szórészlet(ek) és rövidítések halmazaira korlátozódik. Az ilyen típusú feladatokat mintaillesztés segítségével lehet értékelni, figyelembe véve a helyesírási hibák lehetőségét.

    - *Felsorolás*: A felsorolásos feladatok során a válaszoló tetszőleges sorrendben és megfogalmazásban sorolja fel a válaszokat ahelyett, hogy a szavakat az előre meghatározott helyekre begépelné. Ebben az esetben a válasz nem feltétlenül csak egyetlen szóból áll. A felsorolásos feladatok értékelése a mintaillesztésre vezethető vissza, mivel nem kell vizsgálni a felsorolt elemek közötti összefüggéseket.

    - *Rövid szöveges válasz, Esszé*: Rövid szöveges válaszok és esszé típusú feladatok esetén a válaszoló saját szavaival, szabad szöveges választ adhat a feltett kérdésre. Automatizált feldolgozásuk egyik módszere a kulcsszókeresési technika, ahol a választ jónak jelöli, ha a kulcsszavak szerepelnek a válaszban, egyébként hamisnak. Ez a módszer álpozitív eredményeket okozhat, mivel a szavak közötti kapcsolatok módosíthatják a mondat jelentését. Ezért ez a megoldás számos esetben nem alkalmazható. A kulcsszóalapú technikák azonban jó kiindulópontot jelenthetnek a feldolgozáshoz, kiegészítve a mondat megértésének és feltérképezésének funkcionalitásával, az értékelés jó eredményeket hozhat.

Egy rövid szöveges válasz azonosításához elengedhetetlen a természetes nyelvi részfeladatok közötti különbségek meghatározása. A kihagyásos és a felsorolás típusú feladatok könnyen felismerhetők a kérdés jellege alapján, de az esszé és a rövid szöveges válaszok határai elmosódottak. A részfeladattípusok közötti különbségek meghatározásához a következő tulajdonságok elkülönítése segíthet:

- hossz,
- fókusz,
- nyitottság.

A **3.1.1.** táblázat összefoglalja a tulajdonságokat és az alfeladattípusok kapcsolatait.

A hossz tulajdonság egyetlen szóra vagy kifejezésre korlátozódik a kihagyásos és a felsorolásos válaszok esetében, míg az esszében néhány bekezdéstől kezdődően egészen több oldal terjedelmű is lehet. A kettő között van a rövid szöveges válasz, ahol a válasz terjedelme egy szótól egy bekezdésig terjedhet.

3.1.1. táblázat. *Természetes nyelvű feladatok tulajdonságai.*

	Felsorolás	**Rövid szöveges válasz**	**Esszé**
Hossz	szó/szavak	pár szótól egy bekezdésig	egy vagy két bekezdéstől néhány oldalig
Fókusz	szavak és szinonimák	tartalom	stílus
Nyitottság	fix	zárt	zárt

A fókusz határozza meg, hogy a válasz mely elemeit és attribútumait értékelik a feldolgozás során. Ebben az esetben a rövid szöveges válaszokban a tényleges tartalom kerül előtérbe, míg az esszében a hangsúly inkább az írásmódon van, mint a szakmai tartalom tömör megfogalmazásán. Rövid szöveges válaszok esetén a fókusz szűkíthető valamilyen részlet elvárásával vagy annak kizárásával.

Az utolsó jellemző a kérdés nyitottságát jellemzi. Egy kérdést nyitottnak tekintünk, ha a válasz példákat és véleményeket tartalmaz, ebben az esetben az ismeretlen fogalom parafrázisként fogalmazható meg. A kérdés zárt, ha a válasz kizárólag tényekre támaszkodik, rövid, értelmes és nem hosszú állításokat tartalmaz. A két definícióból következik, hogy a rövid szöveges válaszok zártak, azaz objektívek, nem tartalmaznak felesleges leírásokat, az esszék pedig nyitottak, példákat és véleményeket fogalmaznak meg.

Ezek a tulajdonságok megkönnyítik a rövid szöveges válaszok értékelését, de szem előtt kell tartani, hogy a válaszolók nem feltétlenül fogják ezeket a

tulajdonságokat követni a saját válaszaikban.

3.1.2 A szakirodalomban elterjedt módszerek

Szöveges válaszok feldolgozására az idők során számtalan rendszert dolgoztak ki, melyeket az alábbi kategóriákba lehet sorolni:

- *Szabály alapú rendszerek.* Legkorábbi kiértékelési megoldások, ahol előre definiált kulcsszavakat, kifejezéseket keresnek a válaszban valamilyen struktúra alapú leírásnak megfelelően. Ilyen elven működik az eMax rövid szöveges válasz kiértékelő modulja, vagy a Moodle beépített rövid válasz modulja.
- *Gépi tanuláson alapuló rendszerek.* Olyan modellek csoportja, melyeket előzetesen tanítani szükséges a helyes válaszokkal. Képesek általánosítani, felismerni a helyes válasz különböző megfogalmazásait. Jó példa a gépi tanuláson alapuló rendszerre az Nottingham-i Egyetemen fejlesztett Project AutoMark.
- *Hibrid rendszerek.* Ezek a rendszerek jellemzően az előbbi kettő kategória előnyeit próbálják ötvözni az eredményesebb kiértékelés érdekében. Jó példa a Turnitin Revision Assistant, melyet főkent az amerikai középiskolákban alkalmaznak.

3.1.2.1 eMax – Intelligent Short Text Assessment

Az eMax [79] ezen modulja egy olyan intelligens oktatási eszköz része, amely a diákok rövid válaszait elemzi, majd automatikusan pontozza azokat, ezzel támogatva a tanárok munkáját gyorsabb értékelést biztosítva. A rendszer működése a formális szemantikai modellre és a szabályalapú kiértékelésre épül. Először a diák válaszát nyelvtanilag és szemantikailag elemzi, majd egy előre definiált "answer space"-hez hasonlítja, amely tartalmazza a helyes vagy elfogadható válaszok mintáit és szerkezetét. Ez a megközelítés lehetővé teszi, hogy a rendszer ne csak a kulcsszavakat, hanem a válaszok jelentését és logikai felépítését is értékelje.

A rendszer előnye a nagy pontosság és a konzisztencia, valamint az, hogy képes kezelni bizonyos mértékű fogalmazásbeli különbségeket. Ugyanakkor az előkészítés és a szabályok kialakítása időigényes folyamat, ami korlátozza a rendszer skálázhatóságát más tantárgyakra vagy témakörökre.

3.1.2.2 Moodle – Short Answer module

A Moodle "Short Answer" [50] kérdéstípus kulcsszóalapú egyezéses megközelítést alkalmaz: a tanár előre megad egy listát a helyes válaszvariánsokról (szóalakok, kifejezések), majd ezek alapján a rendszer ellenőrzi a diák beviteli mezőjét. Ha a válasz megfelel az egyik mintának (kulcsszó + lehetséges variációk), automatikusan adható a teljes vagy részleges pontszám. A formai leírás szabályalapú, nem igényel mély nyelvi elemzést vagy szemantikai értelmezést.

A rendszer fő előnye a gyors automatizálható javítás, amelynek köszönhetően nagy számú válasz is könnyen értékelhető azonnal a tanár beavatkozása nélkül. Egyszerű és kiszámítható logikát használ, ami megbízható eredményt ad, így a diák gyors visszajelzést kap, ami formálisan is segíti a tanulási folyamatot. Azonban a kulcsszó alapú rendszer egyik gyengesége, hogy nem rugalmas a szinonimák, eltérő megfogalmazások vagy helyesírási eltérések kezelésében. Csak az előre definiált formákat ismeri fel, így, ha a diák egy helyes, de máshogyan kifejezett választ ad, a rendszer elutasíthatja.

3.1.2.3 Project AutoMark - Automatic Short Answer Marking

A Project AutoMark [51] rövid, szabad szöveges válaszok automatikus pontozására épül, főként biológiai kérdésekre fókuszálva. A rendszer shallow processing [52,53] (sekély feldolgozási) módszert használ, nem mély nyelvi elemzést: a válaszokat részleges szintaktikai címkézéssel elemzi, majd kulcsszavak és sablonok alapján információ kinyeréssel dolgozik. A sablonminták (pl. „fertilised egg” + „split” + „into two”) 200 ember által pontozott válaszmintából származnak, és kézzel létrehozva segítik a variánsok felismerését (pl. „split”, „has split”, „splitting”).

A használt modell előnyei közé tartozik a robusztusság a helyesírási hibákkal szemben, mivel a feldolgozási módszer következtében nem igényli a nyelvi elemek minden aspektusának megértését. Hátrányai között megemlíthető az időigényes sablonok kialakítása, illetve az ellentmondások (például a kettős tagadás) kezelésének hiánya, pont a mélyebb nyelvi feldolgozás hiánya miatt.

3.1.2.4 Turnitin Revision Assistant

A Turnitin Revision Assistant [54] rendszere egy gépi tanuláson alapuló, valós idejű visszajelzést biztosító, trait-rubrika (analitikus értékelőrács) alapú írássegítő eszköz, mely lehetővé teszi a diákok számára, hogy folyamatosan értékelést kapjanak dolgozatukról. A diák igényelhet „Signal Check”-et – mely négy kulcsfontosságú dimenzió szerint (fókusz, bizonyítékhasználat, szervezettség és nyelvezet) jeleníti meg a teljesítményét. A rendszer ezután maximum négy mondatot emel ki a dolgozatból, és részletes, célzott javaslatokat nyújt a javításhoz.

A rendszer nagy előnye az azonnali visszajelzés lehetősége mellett a gépi tanulásnak köszönhető rugalmas felhasználása. Hátránya a teljes válasz értékelésének hiánya, az továbbra is a tanárra hárul.

3.1.3 Saját módszer bemutatása

A rövid szöveges válaszok automatikus feldolgozási módszerei két kategóriába sorolhatók: *kulcsszóalapú módszer* és *tartalomfelismerés*. A választott módszer nagyban befolyásolja a kifejlesztett rendszer felépítését és működését.

A kulcsszóalapú módszerrel a kérdés készítőjének meg kell adnia a kulcsszavak azon halmazát, amely tartalmazza a helyes, helytelen és várt kulcsszavakat. A rendszer mintaillesztés segítségével értékeli ki a hallgató válaszát ezen kulcsszavak alapján, figyelembe véve a rövidítéseket és a helyesírási hibák lehetőségét. Ezek közül a technikák közül a legösszetettebbek azok, amikor nemcsak a hibás karakterek számát maximalizálják a szóban (a szó hosszától függően), hanem a szó hosszának különbsége is befolyásolja a javítás folyamatát. Általánosságban elmondható, hogy a válaszok tisztítása és javítása a szógyakorisági szótár segítségével történik, amelyet olyan algoritmusok egészítenek ki, amelyek képesek a helyesírási hibák észlelésére. Ezen szótárak használatához a bemeneti szó karaktereinek összes permutációját kell generálni, amelyekből a szótár a gyakoriság alapján meghatározza, hogy mi a „legmegfelelőbb” szó. Sajnos ennek a módszernek a használata rendkívül költséges, és az eredmény nem garantált, különösen azokban az esetekben, amikor a bemeneti szó nincs a szótárban.

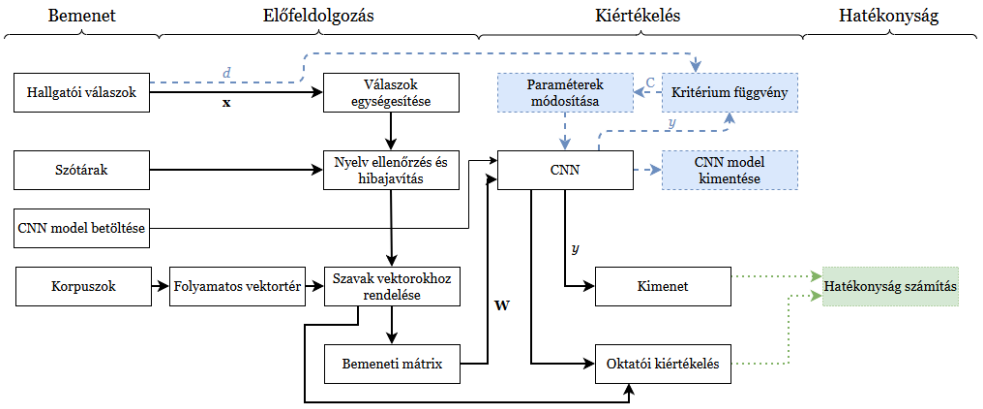
A tartalomfelismerés során a válasz mögöttes jelentését és tartalmát próbáljuk kiértékelni, figyelembe véve a helyesírási hibák lehetőségét, bár ezek a hibák kevésbé befolyásolják a feldolgozást. Ebben az esetben a válasz nem szavak halmaza, amelyeket elemenként kell feldolgozni, hanem egy szorosan összekapcsolódó gráf csomópontjai. Tehát az algoritmusnak meg kell próbálnia megérteni a választ, mivel a szavak (vagy tagmondatok, mondatok) sorrendje hatással van és módosítja a mondat fogalmát.

A két módszer önmagában is alkalmazható rövid szöveges válaszok értékelésére, de kombinálható is. Függetlenül attól, hogy melyik modellt választjuk, a rövid szöveges válasz értékeléséhez a következő négy fontos szempont szükséges:

- kulcsszavak és szinonimák meghatározása;
- kulcsszavak közötti kapcsolatok értékelése;
- szövegmódosító hatások felismerése;
- az értékelt hallgató válaszának összehasonlítása az oktató válaszaival.

3.1.3.1 A rendszer modellje

A módszer egy konvolúciós neurális hálózaton alapuló tartalomfelismerés, amelyet adathalmaz előkészítés, előfeldolgozás és hatékonyság vizsgálat egészít ki, ahogy az a **3.1.1.** ábrán is látható. Ezen az ábrán a kék szaggatott vonallal jelölt elemeket csak a betanítási fázisban, a zöld pontozottal jelölt elemeket pedig csak a modell hatékonyságának meghatározására használjuk.



3.1.1. ábra. *Rövid szöveges válaszok feldolgozásának modellje.*

3.1.3.1.1 A bemenet

A neurális hálózat [55,56] legfontosabb eleme a konzisztens tanulóválaszok halmaza, amelyek segítségével beállítható a hálózat várható működése. A tanulóválaszok halmazán kívül szükség lesz egy validációs válaszkészletre is, amelyek ellenőrzik a tanulófolyamat helyességét, elkerülve a túltanulást és a túlillesztést. Mindkét halmaz vegyes pozitív és negatív mintákat tartalmaz, beleértve a várható kimenetet is, de a modell használatakor a bemeneti válaszok nem tartalmazzák a kimeneti értékeket, azokat a modell állítja elő, és szükség esetén az oktató ellenőrizheti azokat.

A bemeneti konverzió a folyamat része, amely a hallgatói válaszokat egységes adattárolási formává alakítja. Az írásos válaszok (amelyek származhatnak papír alapú kérdőívből vagy elektronikus tesztrendszerből) egységesítésére van szükség, hogy mind a gép, mind az ember könnyen megértse azokat. A kiválasztott tárolási módszer ezért a CSV (Comma Separated Values) által meghatározott formátum, ahol az adatok (válasz és a hozzá tartozó pontszám) soronként szervezettek. A tárolás előnye, hogy az elválasztó karaktereknek köszönhetően az algoritmikus feldolgozás könnyen elvégezhető, és a fájl a modelltől függetlenül olvasható és szerkeszthető.

3.1.3.1.2 Adathalmaz előkészítés

A természetes nyelv neurális hálózatokkal történő feldolgozásának lényegi kérdése a bemeneti adatok formája. Vegyük például a következő mondatot, amelyet fel akarunk dolgozni:

"Ha a cikk romlandó, akkor FIFO egyébként LIFO kiszolgálási politikát célszerű követni."

A példában a helyesen írt szavak mellett írásjelek, rövidítések és helytelenül írt szavak is találhatók. Ha ezt a választ előfeldolgozás nélkül értékeljük ki, a hálózat nem lesz képes helyesen kiértékelni, mivel a modell a helytelenül írt szavak miatt nem tudja azokat vektorokhoz párosítani. Az előforduló problémák, amelyekre fel kell készülni:

- Ha a szót írásjel előzi meg vagy követi (egységesítés szükséges);
- Ha a szó el lett gépelve (nyelvi javítás szükséges);
- Ha a vektortér nem tartalmazza a szót (vektortér rekonstrukciója

szükséges);

- Ha a szónak más alakja is van a válaszban (további szótárak és korpusz szükséges).

A felsoroltakon túl további váratlan esetek is előfordulhatnak a modell előhívás fázisa során (például a nagybetűk esete), amelyekre a modellnek fel kell készülnie. A válaszok átalakítását körültekintően kell elvégezni. Ha a rendszer rugalmatlan, akkor ezekben az esetekben a szavak vektorai nem találhatók meg (mivel nem léteznek), így a válasz feldolgozása megszakad. Ha azonban a modell túl rugalmas (például az algoritmus a „FIFO" helyett a „first-in FO" szót is elfogadja), akkor a válasza az előfeldolgozás során sérülni fog (vagy a rossz válasz is el lesz fogadva).

A válasz egységesítésekor a felesleges írásjeleket eltávolítjuk, ami csökkenti a kapott vektortér méretét. A példamondatot megvizsgálva látható, hogy a jelentése szempontjából irreleváns, hogy a vessző a mondatban a szó után szerepel-e: „romlandó**,**" vagy sem: „romlandó". A fenti példa esetében a vektortérben mindkét verziónak meg kellene jelennie. A szavak valódi jelentését azonban a szavak kontextusa fogja meghatározni, amelyet a konvolúciós neurális hálózat (CNN) megtanul felismerni és értelmezni. [57]

A probléma (szóvektor-leképezés) megoldásának egyik kritikus pontja a szavak nyelvtani helyességének ellenőrzése. Mivel a helytelenül írt szavak nem találhatók meg a vektortérben, így algoritmikusan a feldolgozásuk sikertelen lesz. A helyesírás-ellenőrzés és javítás a Google Fordító által használt javító mechanizmuson [58] alapul. A módszer jellemzője a gyakorisági szótár, amely a szavakat és azok átlagos megjelenését tartalmazza. A modellben a következő gyakorisági szótárak kerültek felhasználásra:

- Általános gyakorisági szótár, amely a nyelv általános szavainak gyakoriságát tartalmazza;
- Kiegészítő gyakorisági szótár, amely a feladatra/szakterületre jellemző szavakat tartalmazza;
- Szinonima szótár, amely az egyes szavak szinonimáit tartalmazza (sok esetben nem lényeges a kifejezés fajtája, így például first in first out" vagy „FIFO" forma is megfelelő a válaszban, de a feldolgozás

szempontjából előnyösebb elkerülni a redundanciát);

- Egy speciális szimbólumlista, amely azokat a szimbólumokat tartalmazza, amelyeket a szavak feldolgozása előtt törölni kell a válaszból (például: *).

A javítás során a (formázott, írásjelekkel ellátott) választ szavakra bontjuk, majd első lépésként a kifejezéseket a szinonima szótár szerint egyesítjük. Ezt követően a feldolgozás végigmegy a válaszban szereplő összes szón, és ellenőrzi, hogy szerepel-e a szógyakorisági szótárakban vagy sem. Ha megtalálja, feltételezi, hogy a szó formátuma megfelelő, és a feldolgozása befejeződik. Ha nem található, akkor új szavakat hoz létre a szóban lévő karakterekből karakterek felcserélésével, permutálásával, kiegészítésével vagy törlésével. A generálás végén a generált szókészletet a gyakorisági szótár segítségével rendezi, és a halmazból kiválasztja a legmagasabb értékű szót. Ha ez a folyamat sikertelen, akkor az eredeti szó érintetlen marad a válaszban. [58]

Ennek a módszernek a hátránya, hogy ha a gyakorisági szótár nem tartalmaz egy szót, de tartalmaz egy hasonlót, akkor az eredetileg helyes szó a javítás során egy rossz szóval lesz helyettesítve.

A helyesírás-ellenőrzés és -javítás olyan mondatot hoz létre, amely nem torzítja a mondat eredeti jelentését és alkalmas neurális hálózati feldolgozásra.

3.1.3.1.3 Vektortér felépítése

Egy jó vektortér létrehozásának feltétele a releváns korpuszok használata, amelyek amellett, hogy minden releváns szót tartalmaznak, a megfelelő mennyiségi és minőségi kapcsolatokat is reprezentálják. Korpuszok esetén az uniformizálás folyamata is ajánlott, ezáltal csökkentve a vektortér méretét. [57]

A vektorteret a word2vec programmal készítettem, ahol a CBOW és a Skip-Gram vektortér-építési technikák is elérhetők. [59] A tér felépítéséhez a CBOW-t használtam, mivel a meglévő korpuszok kisebb halmazt alkottak [60] (nagyobb korpuszok esetén a Skip-Gram modell használata javasolt). Végül létrehoztam a vektorteret, amely felhasználható a hallgatói szöveges válaszok mátrixszá konvertálására. Ez a mátrix betölthető a neurális hálózat

bemenetére.

A helyes mátrix létrehozásához a válasz minden szavát meg kell találni a vektortérben, ami egy egyszerű kapcsolat lesz a szavak és a vektorok között. Probléma akkor merül fel, ha egy vagy több szó nem található meg a térben. Ez akkor fordulhat elő, ha a válasz uniformizálása helytelen volt, vagy ha egy olyan szó szerepelt a válaszban, amelyet a korpusz nem tartalmazott. Ebben az esetben a lehetőségek a következők:

- A szót figyelmen kívül hagyják, tudván, hogy a válasz torz lesz, vagy értelmetlen (nem ajánlott);
- A választ további feldolgozás nélkül továbbítják az oktató értékeléséhez (ajánlott a betanított rendszer számára az előhívási fázisban);
- Ha a szó jelentéssel bírt, akkor áttekintjük a korpuszt, a vektorteret és a válasz uniformizálásának folyamatát (a modell betanítása során ajánlott).

Miután megtaláltuk a válasz szavaihoz tartozó vektorokat, előállítható a mátrix, ahol a sorok a szavak vektorainak reprezentációi, az oszlopok pedig a vektorok elemei. A vektorok, azaz a mátrix oszlopainak hossza a vektortér létrehozása során határozható meg. A szavak, azaz a mátrix sorainak számát azonban előre rögzíteni kell, aminek eredményeként vagy a fennmaradó sorokat jelentés nélküli vektorokkal kell kitölteni, vagy a hosszabb válaszokat le kell vágni. Az eddigi tapasztalatok (és a rövid szöveges válasz jellege) alapján kijelenthető, hogy 100 szónak elegendőnek kell lennie a válasz megfogalmazásához, amely érték a feladattól és a betanítás eredményétől függően állítható, változtatható.

A válasz mátrix esetén a tárolás kérdéses lehet, ami nem szükséges, mivel az abban található információ megtalálható a szöveges válaszban és a vektortérben is. Miután előáll a bemeneti mátrix, elkezdődhet a kiértékelés a CNN segítségével.

3.1.3.1.4 Osztályozás

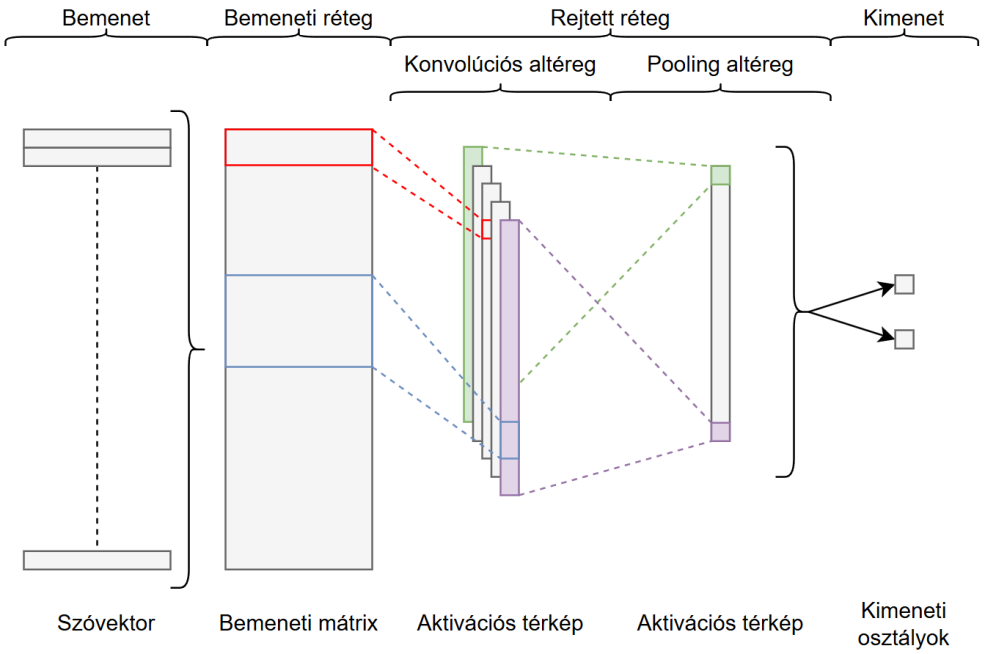
A feldolgozás két részre osztható attól függően, hogy a cél a hálózat betanítása vagy használata. Ha a cél a betanítás, akkor a bemeneti adatokat

tanuló- és validációs halmazra kell szétválasztani, és a hálózat paramétereit a kritériumfüggvény és az oktató értékelése alapján kell beállítani. Ha egy már betanított neurális hálózatot szeretnénk használni, akkor azt be kell tölteni a modellbe, majd a válaszokat át kell adni a kimeneti értékek kinyeréséhez (a kiértékelési folyamat frissíti a bemeneti CSV fájl tartalmát a válaszok pontszámával).

A tanító adatok kiválasztásakor célszerű az alábbiak szerint eljárni:

- a válaszok természetüknél fogva a lehető legnagyobb mértékben különbözzenek;
- a legtöbb (de nem az összes) szó szerepeljen a kiválasztott készletben;
- a kiválasztott készlet tartalmazzon közel azonos számú negatív és pozitív mintát.

A természetes nyelv feldolgozása során a CNN olyan konvolúciós szűrőket használ, amelyek jellemzően a mátrix teljes sorát (az egész szót) lefedik, ellentétben a képfeldolgozással, ahol a szűrők két vagy három képpontot fednek le. Vagyis a szűrő szélessége általában megfelel a mátrix szélességének, a magassága pedig változó, jellemzően kettő-öt szó, ahogy az a **3.1.2.** ábrán is látható. [61,62]



3.1.2. ábra. *A CNN felépítése természetes nyelvű feladatok feldolgozása során.*

A gépi látás számára hasznos intuíciók (invariancia, kompozíció) nem lesznek hasznosak a természetes nyelv feldolgozásakor. A szavak kompozíciója meghatározó lehet alacsony szinteken (igék, végződések stb.), de a magasabb absztrakciós szinten nem segíti a feldolgozást, ellentétben a képfeldolgozással. Ezért a feldolgozás során a szavak jelenlétére vagy előfordulási helyére koncentrálunk.

A CNN hálózat két rejtett réteget tartalmaz. Az első a konvolúciós réteg, amelynek feladata a válaszban található lényeges információk kiemelése. A második az összevonó réteg, amely a konvolúciós réteg kimenetét egyetlen értékké összesíti.

A konvolúciós rétegben a következő képletet használjuk az *i*-edik kimenet meghatározására:

$$C_i = \sum_{j=1}^{m} \sum_{k=1}^{n=100} \left(\boldsymbol{S}_{[i-m+1:i]} \circ \boldsymbol{F} \right)_{j,k} \qquad (3.1.1)$$

ahol $\boldsymbol{S}$ a bemeneti mátrix (a válasz része, 2-5 szóból áll) 0-val kiegészítve a mátrix tetején és alján, *[i-m+1:i]* mátrix sorainak megszorítása *i-m+1* és *i* közé; $\boldsymbol{F}$ a szűrő mérete *m* magassággal, ° a mátrixok Hadamard-féle szorzata; $\boldsymbol{C}_i$ a kimeneti mátrix *i*-edik komponense.

A pooling rétegként a maximális pooling réteget választottam, mivel gyors és segítségével kiemelhetők a szöveg fontosabb jellemzői, továbbá segít abban, hogy a hálózat invariáns legyen a kisebb eltérésekre. Hátránya azonban, hogy a hálózat könnyen túltanulhatja magát. A rétegben a következő általános képlet használható:

$$\boldsymbol{C}_{pooled} = \begin{bmatrix} pool\big(\alpha(\boldsymbol{C}_1 + b_1 * \boldsymbol{e})\big) \\ \dots \\ pool\big(\alpha(\boldsymbol{C}_n + b_n * \boldsymbol{e})\big) \end{bmatrix} \tag{3.1.2}$$

$$pool(x) = \max(0, x) \tag{3.1.3}$$

ahol a $pool$ a max pool művelet; $\alpha()$ az aktivációs függvény (az összevonás küszöbértéke); $\boldsymbol{C}_i$ a konvolúció kimeneti mátrixának i-edik komponense, $\boldsymbol{C}_{pooled}$ pedig az összevonás kimenete (ebben az esetben egy egyszerű szám).

Végül a hálózat egy teljesen összekapcsolt réteget tartalmaz, amely egy egyszerű, előrecsatolt neurális hálózat SoftMax aktivációs függvénnyel. Ez a réteg valószínűségeloszlás alapján osztályozza a bemenetét az alábbiak szerint:

$$p(y = j|\boldsymbol{x}) = \frac{e^{\boldsymbol{x}^T \theta_j}}{\sum_{k=1}^{K} e^{\boldsymbol{x}^T \theta_k}} \tag{3.1.4}$$

ahol θ a k-adik osztály súlyvektora. A képlet kimenete egy olyan vektor lesz, amelynek annyi eleme van, ahány osztályt szeretnénk létrehozni, és az elemeik értékei az osztályba tartozás valószínűségei lesznek. [63,64]

Minden válaszhoz tartozik egy kételemű kimeneti vektor (a jó és a rossz válasz valószínűsége). A kimeneti vektor alapján meg kell határozni, hogy a válasz jó vagy rossz volt-e. A bemeneti válaszok túlnyomó többsége nem sorolható egyértelműen egyetlen osztályba, ezért kompromisszumot kell kötni a válasz jóként vagy rosszként való elfogadásakor. Tegyük fel például, hogy van egy betanított hálózat, amely a bemeneti válaszok 42%-át a „rossz válasz” osztályba, 58%-át pedig a „jó válasz” osztályba sorolja. A kimeneti réteg Softmax aktivációs függvénye által szolgáltatott valószínűségi értékek feldolgozásakor eltértem a hagyományos, maximális valószínűségen alapuló osztályozástól. Míg az alapértelmezett megközelítés bármilyen 0,5 feletti

értéket elfogadna az adott osztályba tartozás bizonyítékaként, jelen kutatásban egy szigorúbb, 0,75-ös küszöbértéket vezettem be. Ez a döntés a pontosság (accuracy) növelését szolgálja a visszahívás (recall) rovására, biztosítva, hogy csak a nagy bizonyossággal „helyesnek” ítélt válaszok kerüljenek elfogadásra. Ennek következtében az előbbi példát nem soroljuk a jó válaszok közé, annak ellenére sem, hogy a modell nagyobb valószínűséggel tekinti azt jónak.

Az itt megadott 0,75 kimeneti küszöb érték empirikus úton került megválasztásra, figyelembe véve a vizsgált rövid szöveges válaszokat, illetve, hogy a kiértékelés szempontjából sokkal károsabb egy rossz szöveget jónak jelölni (például a válaszban az elvárt szavak összefüggéstelen egymásutániságát is elfogadni), mintha egy jó szöveget (például helyesírási hibákkal teli választ) jelölnék rossznak. Természetesen a küszöb érték egy ennél pontosabb meghatározásához további vizsgálatok szükségesek, amivel a jelentkező veszteség tovább minimalizálható az adott bemeneti kategóriák értékeinek jellegét figyelembevéve.

3.1.3.1.5 Hatékonyság

A modell hatékonyságának vizsgálatakor feltételezzük, hogy a hálózat minden kérdésre a helyes választ adja. Vagyis a modellen az összes rendelkezésre álló adat áthalad, miután a szükséges szótárak, a vektortér és a betanított modell betöltődik. A kiértékelés végén a modell a következő kategóriák szerint jelöli a válaszokat:

- Oktatói értékelés szükséges;
- A kérdésre adott válasz helyes;
- A kérdésre adott válasz helytelen.

A kapott eredmények összehasonlításra kerülnek az oktató által megadott eredményekkel, ami meghatározza, hogy hány választ értékelt ki helyesen vagy helytelenül a rendszer, és hányat nem lehetett vele kiértékeltetni. Ez eredményezi a modell azon százalékos arányát, amely a feldolgozási hatékonyságot jellemzi.

3.1.4 A bemutatott módszer értékelése

A modell teszteléséhez angol nyelvű SMS-eket [65] választottam, amelyek

tartalmuk alapján két kategóriába lettek sorolva (ham, spam), valamint angol nyelvű fórumbejegyzéseket [66], amelyeket érzelmi tartalmuk alapján szintén két csoportra (pozitív, negatív) lettek bontva. A két adathalmaz főbb jellemzőit a **3.1.2.** táblázat mutatja. Mindkét adathalmaz esetén a válaszok közel 50%-a lett kiválasztva a modell tanításához. Az angol nyelvű adathalmaz értékelésének eredményeit a **3.1.3.** és a **3.1.4.** táblázat, illetve a **3.1.3.** ábra foglalja össze.

A **3.1.4.** táblázatban szereplő igazságmátrixokból számított értékeket **3.1.5.** táblázat tartalmazza. Az SMS-ek és Forum posztok validációs adathalmazai esetén a modell kiemelkedő és stabil teljesítményt nyújt, amit a magas, 0,853-as és 0,744-es F1-pontszámok is igazolnak. Az SMS-ek osztályozása mutatja a legkiegyensúlyozottabb eredményt, ahol a valódi pozitívok aránya (VPR = 0,877) és a pozitív prediktív érték (precision, PPE = 0,830) közötti csekély eltérés minimális torzításra utal. Ezzel szemben a Forum posztoknál a PPE (0,724) elmarad a VPR-től (0,766), ami matematikailag a téves riasztások (HP = 40) gyakoribb előfordulását jelzi. Az alkalmazott magas (0,75-ös) küszöbérték ellenére itt már több olyan esetet is "azonosnak" minősít a modell, melyek a valóságban különbözőek, így a kritikus sikerességi mutató 0,593-ra mérséklődik. A kritikus sikerességi mutatón keresztül jól látható (mivel nem veszi figyelembe a valódi negatívok számát), hogy a modell nehezebben képes kezelni a hosszabb és összetettebb mondatszerkezeteket és a Forum posztokra jellemző szubjektív érzelmeket, az iróniát és a szarkazmust.

3.1.2. táblázat. *A tanítás és validálás során használt adathalmazok főbb jellemzői.*

	SMS-ek	**Forum posztok**	**SMS-ek (redukált)**	**Hallgatói válaszok**
Teljes elemszám	1 324	1 000	150	86
Ham/Pozitív elemszám	1 000	500	95	56
Spam/Negatív elemszám	322	500	55	30
Kiegyensúlyozatlanság foka	közepes	kiegyensúlyozott	enyhe	enyhe

Ezt követően a modellt magyar hallgatók válaszaival teszteltem, melyeket az oktató értékelt ki. Mivel ez az adathalmaz sokkal kisebb, az értékelést ismét elvégeztem az angol SMS adathalmaz egy csökkentett halmazán, hogy összehasonlíthassam a különbségeket (az adathalmazok főbb tulajdonságait a **3.1.2.** táblázat mutatja be). Az értékelés eredményeit a **3.1.6.** és a **3.1.7.** táblázat és az **3.1.4.** ábra foglalja össze.

A **3.1.8.** táblázatban szereplő értékek a **3.1.7.** táblázat igazságmátrixainak értékeiből kerültek kiszámításra. A redukált SMS-ek és a Hallgatói válaszok elemzése során a modell prediktivitása drasztikusan lecsökken, amit a 0,500 és 0,593 közötti alacsony pontossági értékek jeleznek, melyek alig haladják meg a véletlen osztályozás szintjét a magas küszöbérték ellenére sem. A redukált SMS-eknél a legalacsonyabb a kritikus sikerességi mutató (0,384), ami az információvesztés miatti bizonytalanságot tükrözi, míg a Hallgatói válaszoknál tapasztalható rendkívül alacsony VPR (0,555) és a viszonylag magasabb PPE (0,666) közötti ellentmondás azt mutatja, hogy a modell ezen a területen túl szigorú, bár ritkán jósol egyezést, a valós pozitív esetek közel felét (HN = 8) képtelen felismerni a választott 0,75-ös hasonlósági küszöb mellett.

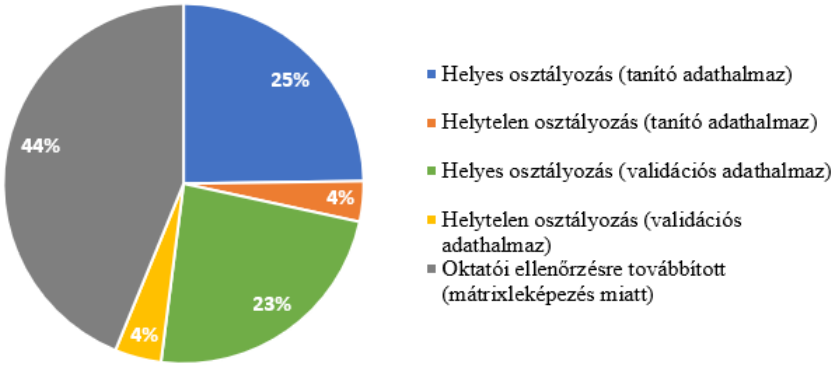
További következtetésként elmondható az eredményekről, hogy a vektortérben lévő szavak erősen befolyásolják a feldolgozható válaszokat, mivel, ha a szó nem található meg a vektortérben, a modell továbbításra kerül az oktatónak kiértékelésre (az SMS-ek és a Forum posztok esetében ez közel 50% (**3.1.3.** táblázat), míg a redukált SMS-ek és a Hallgatói válaszok esetén ez körülbelül 20% (**3.1.5.** táblázat)). Továbbá az adathalmaz mérete erősen befolyásolja a kiértékelés hatékonyságát. SMS-ek esetén az adathalmaz nagyobb mérete lehetővé tette a hálózat számára, hogy „jobban tanuljon” a mintákból, de ezzel ellentétes volt a hallgatói válaszok esetében, ahol a hálózat nem ismerte fel a mintákat. Az eredmények alapján kijelenthető, hogy a modell nagyon érzékeny az adatméretre, de az értékelendő nyelvre nem. A vektortér és a szótárak helyettesítésével a modell alkalmas más nyelveken írt rövid szöveges válaszok kiértékelésére is.

3.1.3. táblázat. *A modell által kiértékelt angol nyelvű adathalmazok eredményei*

	SMS		**Fórum poszt**		***Átlag***
Elemek száma az adathalmazban	1324	100%	1000	100%	*100%*
A válasz az „Oktatói ellenőrzés”-re hivatkozik (mátrixtérképezés miatt)	604	45.6%	519	51.9%	*48.7%*
Tanításra alkalmas elem	720	54.4% (100%)	481	48.1% (100%)	*51.3% (100%)*
Tanító adathalmaz mérete	364	100%	231	100%	*100%*
Helyes osztályozás	317	87%	224	97%	*92%*
Helytelen osztályozás	47	13%	7	3%	*9%*
Validációs adathalmaz mérete	356	100%	250	100%	*100%*
Helyes osztályozás	302	85%	178	71%	*79%*
Helytelen osztályozás	54	15%	72	29%	*21%*

3.1.4. táblázat. *A modell által kiértékelt angol nyelvű validációs adathalmazok igazságmátrixa.*

SMS		Számított hasonlósági érték szerint			
		>= 0,75		< 0,75	
Megoldott feladat szerint	Azonos	*VP*	**157**	*HN*	**22**
	Különböző	*HP*	**32**	*VN*	**145**
Forum poszt		Számított hasonlósági érték szerint			
		>= 0,75		< 0,75	
Megoldott feladat szerint	Azonos	*VP*	**105**	*HN*	**32**
	Különböző	*HP*	**40**	*VN*	**73**



3.1.3. ábra. *A modell által kiértékelt angol adathalmaz összesített eredménye.*

3.1.5. táblázat. *A* ***3.1.4.*** *és a* ***3.1.6.*** *táblázat igazságmátrixaiból számított értékek.*

	Valódi pozitív arány (VPR) $\frac{VP}{P}$	**Pozitív prediktív érték** (PPE) $\frac{VP}{VP+HP}$	**Pontosság** $\frac{VP+VN}{P+N}$	**F1 pontszám** $\frac{2*PPE*VPR}{PPE+VPR}$	**Kritikus sikerességi mutató** $\frac{VP}{VP+HN+HP}$
SMS	0,877	0,830	0,848	0,853	0,744
Forum poszt	0,766	0,724	0,712	0,744	0,593

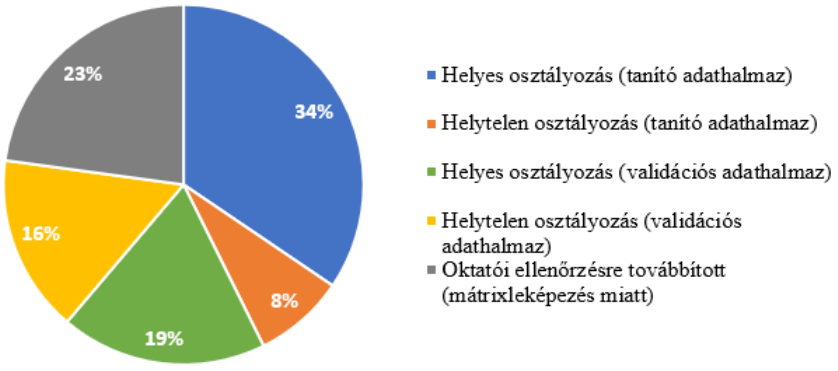
3.1.6. táblázat. *A redukált SMS-ek (angol) és a modell által kiértékelt magyar hallgatói válaszok összesített eredményei.*

	SMS (redukált)		**Hallgatói válaszok**		***Átlag***
Elemek száma az adathalmazban	150	100%	82	100%	*100%*
A válasz az „Oktatói ellenőrzés”-re hivatkozik (mátrixtérképezés miatt)	42	28%	11	13.5%	*20.8%*
Tanításra alkalmas elem	108	72% (100%)	71	86.5% (100%)	*79.2% (100%)*
Tanító adathalmaz mérete	60	100%	39	100%	*100%*
Helyes osztályozás	49	81.6%	31	79.5%	*81%*
Helytelen osztályozás	11	18.4%	8	20.5%	*19%*
Validációs adathalmaz mérete	48	100%	32	100%	*100%*
Helyes osztályozás	24	50%	19	59.3%	*55%*
Helytelen osztályozás	24	50%	13	40.6%	*45%*

3.1.7. táblázat. *A modell által kiértékelt redukált SMS-ek (angol) és a magyar hallgatói válaszok validációs adathalmazok igazságmátrixa.*

SMS (redukált)		Számított hasonlósági érték szerint			
		>= 0,75		< 0,75	
Megoldott feladat szerint	Azonos	*VP*	**15**	*HN*	**11**
	Különböző	*HP*	**13**	*VN*	**9**

Hallgatói válaszok		Számított hasonlósági érték szerint			
		>= 0,75		< 0,75	
Megoldott feladat szerint	Azonos	*VP*	**10**	*HN*	**8**
	Különböző	*HP*	**5**	*VN*	**9**



3.1.4. ábra. *A redukált SMS-ek (angol) és a modell által kiértékelt magyar hallgatói válaszok összesített eredményei.*

3.1.8. táblázat. *A **3.1.4.** és a **3.1.6.** táblázat igazságmátrixaiból számított értékek.*

	Valódi pozitív arány (VPR) $\frac{VP}{P}$	**Pozitív prediktív érték** (PPE) $\frac{VP}{VP+HP}$	**Pontosság** $\frac{VP+VN}{P+N}$	**F1 pontszám** $\frac{2*PPE*VPR}{PPE+VPR}$	**Kritikus sikerességi mutató** $\frac{VP}{VP+HN+HP}$
SMS (redukált)	0,576	0,535	0,500	0,555	0,384
Hallgatói válaszok	0,555	0,666	0,593	0,606	0,434

3.1.5 Konklúzió

A modell teljesítette azt a célkitűzést, hogy rövid szöveges válaszokat lehessen kiértékelni, de a magyar hallgatói válaszok esetében ez az elvárás csak részben teljesült. Ennek oka a modell működéséhez szükséges nagy mennyiségű adat igénye, mely nem állt rendelkezésre.

A saját módszerként ismertetett konvolúciós neurális hálózat alapú kiértékelése a hozzá kapcsolódó szavak vektorizálásával egy jó megoldás lehet abban az esetben, mikor konkrét, jól körülhatárolható szöveget (választ) kell osztályozni, ahol fontos a válasz pontossága. Azonban nem szabad elmenni a tény mellett, hogy a nagy nyelvi modellek folyamatos fejlődése jobb, de összességében erőforrásigényesebb kiértékelési rendszerek megjelenéséhez vezethet.

3.1.6 Tézis kimondása

2.1. tézis – Kidolgoztam egy módszert, amely alkalmas szabadszavas rövid szöveges válaszok kiértékelésére. A módszer az eredetileg képfeldolgozási területen használt konvolúciós neurális hálózatokon alapul, kiegészítve azt speciális elő- és utófeldolgozó lépésekkel, amelyek jelentősen növelik a hálózat hatékonyságát a megadott területen. A valódi adatokon történő vizsgálatok kimutatták, hogy megfelelő méretű tanítási adatbázis megléte

esetén az általam javasolt modell nagy pontosságot ért el. [S4]

3.2 Forráskódok osztályozása a programozási nyelv tokenjeiből épített szóvektortér segítségével

3.2.1 Bevezetés

A fejlesztőknek a kódfájlokat vagy forrásfájlokat funkcionalitásuk szerint kell osztályozniuk (például Maximum kiválasztás, Rendezés stb.), hogy ezek később könnyen kereshetők és használhatók legyenek. A forráskód osztályozásának egy másik aspektusa, amikor sokan oldanak meg egy feladatot, ezáltal közel ugyanazt az algoritmust elkészítve, például egy iskolai vizsgát. [S5] Ilyen helyzetekben gyakori, hogy a forráskód ugyanazon a nyelven, közel azonos szintaxissal készül (bár ezt nagymértékben befolyásolja a fejlesztő tudása és tapasztalata). A cél az ugyanarra a megoldásra készült algoritmusok végrehajtás nélküli pontos meghatározása. Ez az osztályozás hasznos fejlesztés alatt (például kódrészlet ajánlás), vagy amikor forrásfájlokat szeretnénk hozzárendelni a feladathoz (például a megoldásokat a vizsgafeladatokhoz. [67,68]

Ez az osztályozási probléma szöveges dokumentumok feldolgozására vezethető vissza, ha a forrásfájlt szövegfájlnak és nem utasítássorozatnak tekintjük. Ebben az esetben el kell távolítani a felesleges információkat (például a megjegyzéseket), és a szavakat (utasítások, vezérlési szerkezeteket stb.) olyan formára kell alakítani, amely lehetővé teszi a neurális hálózat betanítását és későbbi használatát. Erre a problémára lehet jó megközelítés a konvolúciós neurális hálózat (CNN) használata. Eredetileg képfeldolgozásra fejlesztették ki ezt a hálózatot, de a CBOW és a Skip-Gram technikák megjelenése után a CNN-t sikeresen használják természetes nyelvi feldolgozására is. [69]

A forráskódok osztályozásának célja a feladatok előre meghatározott számú osztályba sorolása. Az összetételükben mutatkozó különbségek ellenére a forráskódok, akárcsak a természetes nyelvek, sajátos struktúrával rendelkeznek (megjegyzések nélkül), amely könnyen azonosítható.

A jelen fejezetben vizsgált, CNN-alapú modellek és a szóvektortér-reprezentációk stabil alapot jelentenek, azonban a 2020 óta megjelent legújabb kutatások már a Transformer-alapú architektúrák és a nagyméretű nyelvi modellek (LLM) felé mozdultak el. Kezdeti megoldások, mint

például a CodeBERT [70] vagy a GraphCodeBERT [71], már nem csupán token-sorozatként, hanem a kód adatfolyam-szerkezetét is figyelembe véve, előtanított modellek segítségével végzik az osztályozást, ami kiemelkedő pontosságot tesz lehetővé kevés tanítóadat mellett is. A legújabb kutatások már olyan modelleket alkalmaznak, mint a StarCoder2 [72] vagy a CodeLlama [73], amelyek több százmilliárd tokenen végzett előtanításuknak köszönhetően mélyebb szemantikai megértéssel rendelkeznek a programozási nyelvek logikai struktúrájáról. A 2024-es és 2025-ös publikációk kiemelik a többfeladatos tanulás és a kódvégrehajtási adatokkal kiegészített reprezentációk jelentőségét; például a legújabb hibrid megközelítések már nemcsak a statikus forráskódot, hanem a futásidejű viselkedési szignatúrákat is integrálják a pontosabb osztályozás érdekében [74]. Ezen túlmenően megjelentek a paraméter hatékony finomhangolási eljárások, mint a LoRA [75], amelyek lehetővé teszik a modellek specifikus kategóriákra való adaptálását minimális erőforrásigény mellett. Ebben fejezetben bemutatott CNN-alapú módszer a legújabb modellekhez viszonyítva egy determinisztikusabb, alacsonyabb számítási kapacitást igénylő és könnyebben értelmezhető alternatívát kínál, amely különösen értékes olyan környezetben, ahol a modell válaszideje és az erőforrás-hatékonyság kritikus szempont.

3.2.2 A szakirodalomban elterjedt módszerek

A forráskódok automatikus osztályozása, hasonlóság szerinti csoportosítása a szoftverfejlesztés és a mesterséges intelligencia határterületén egyre nagyobb figyelmet kap. Az elmúlt években több olyan megközelítés is született, amelyek különféle gépi tanulási modellek (például konvolúciós neurális hálók, rekurzív hálók, vagy vektoralapú beágyazott modellek) alkalmazásával próbálják megérteni és feldolgozni a forráskódok szerkezetét és jelentését.

Ezen módszerek egy része a forráskód szintaktikai vagy szemantikai szerkezetét próbálják feldolgozni, például absztrakt szintaxisfák (AST) segítségével, míg más megközelítések inkább a kódok tokenizált verzióját kezelik természetes nyelvként. A cél mindkét esetben azonos, vagyis, hogy egy géppel tanulható reprezentációja készüljön el a kódnak, amely alapján az osztályozás megvalósítható.

3.2.2.1 Klónozás detektálása mélytanulás segítségével

A „Deep Learning Code Fragments for Code Clone Detection” [76] című tanulmány a kódklónok felismerésének problémáját vizsgálja mélytanulás segítségével, amely automatikusan tanulja meg a kód fragmentek jellegzetességeit, így detektálva a hasonló funkciójú, de akár eltérő szerkezetű forráskódokat.

A bemutatott módszer előnyei közé tartozik, hogy több klóntípust is képes felismerni (1.–4. típus), így nemcsak az egyszerű szöveghasonlóságon alapuló, hanem a szintaktikailag különböző, de funkcionálisan hasonló kódokat is megtalálja. A kutatók 398 fájl és 480 metódus klónpárt vizsgálva 93%-os pontosságot értek el és mutatták ki, hogy számos olyan klónpárt is azonosítanak, amelyeket a Deckard [77] nevű hagyományos eszköz nem, vagy csak részben ismer fel. A módszer hátrányai azonban nem elhanyagolhatók. Egyrészt a tanításhoz nagy mennyiségű annotált klón szükséges, amely gyűjtése és előkészítése számításigényes. Másrészt a neurális hálók betanult jellemzői kevésbé átláthatók az ember számára, így a döntéshozatal magyarázata komplex, bizonyos esetekben nem is lehetséges. Emellett a kutatás Java nyelvre fókuszál, így más programozási nyelvek vagy kódstílusok esetén a teljesítmény csökkenhet, és további újratanításra lehet szükség.

3.2.2.2 Code2Vec

A Code2Vec [78] egy újfajta kódreprezentációs módszert vezetett be, amely célja, hogy gépi tanulási algoritmusok számára tanulható, kompakt és informatív vektorábrázolást biztosítson. A szerzők kiindulópontja az volt, hogy a hagyományos token alapú megközelítések nem képesek jól megragadni a kód szemantikai struktúráját. Ezért az absztrakt szintaxis fából építették fel a vektorteret, amely fő újítása a fában lévő útvonalak halmazként való reprezentációja volt.

Előnyei között említhető, hogy a modell nem csak felszínesen képes elemezni a forráskódokat, hanem az útvonalakon keresztül a kódelemek közötti kapcsolatok mélyebb szerkezetét is figyelembe tudja venni, így jobban képes általánosítani különböző, de funkcionálisan hasonló megvalósítások esetén. További előnye, hogy programozási nyelvtől

független, így elméletileg bármely programozási nyelv esetén alkalmazható, amennyiben rendelkezésre áll annak absztrakt szintaxis fája. Ugyanakkor hátrányai között megemlíthető, hogy jelentős előfeldolgozási lépésekre van szükség, amelyek nem triviálisak: az AST-k generálása, az útvonalak feldolgozása számottevő számítási igénnyel jár. Emellett a modell eredetileg metódusnév predikcióra lett optimalizálva, így más osztályozási feladatok esetén csak korlátozottan alkalmazható.

3.2.3 Saját módszer bemutatása

3.2.3.1 Adatok előkészítése

A neurális hálózatokra jellemző, hogy nagy mennyiségű, kiváló minőségű adatra van szükség a betanításhoz. Ennek a követelménynek a teljesítéséhez a használt forráskódoknak teljes mértékben meg kell oldaniuk a feladataikat. Forráskódok esetében ezt az elvárást az egységtesztek (amelyek ellenőrzik, hogy a kód egyes részei (általában függvények) a várt módon működnek-e) és különböző metrikák (futási idő, memóriahasználat) mérése biztosítja. [79]

A modell betanításához használt adathalmazt a HackerRank webhelyéről származó több ezer megoldásból gyűjtöttem össze. A feladatok kiválasztása a komplexitás, a hasonlóság és az azonos nyelven elérhető megoldások száma alapján történt.

Végül három feladatot választottam ki, és töltöttem le a megoldásaikat (C#, Java, Python2 nyelven). Az „A very big sum” [80] és a „Simple array sum” [22] feladatok közel azonos megoldásokat igényelnek, így alkalmasak a modell osztályozó képességének tesztelésére. A harmadik feladat (Big sorting [81]) jelentősen eltér ezektől, ezért megoldásai szintén hozzájárulnak a hálózat minőségének javításához a betanítási fázisban. A különböző programozási nyelvek választását a feladatonkénti, nyelvenkénti különböző megoldások száma és az eltérő implementációtípus indokolta. Kezdetben Java feladatokkal foglalkoztam, hogy megbizonyosodjam arról, hogy elegendő forrásfájl áll-e rendelkezésre a teszteléshez. Később (amíg a modell fejlesztés alatt állt) csökkentettem a tesztadatok számát, és megváltoztattam a feladat megoldási nyelvét, ahol kevesebb megoldást találtam (például C#).

A **3.2.1.** táblázat a begyűjtött feladatok és megoldásaik eloszlását mutatja programozási nyelvek szerint. Egyszerű HTTP kéréseket használtam a megoldások letöltésére, a rossz kéréseket (ahol a forráskódminták fordítása és futtatása sikertelen volt) figyelmen kívül hagytam. Összesen 365 141 forráskódfájlt sikerült letölteni. [S6]

3.2.1. táblázat. *A begyűjtött feladatok programozási nyelvenkénti száma.*

Feladat	**Programozási nyelv**	**Letöltött megoldások száma**
	C#	40 417
A very big sum	Java	86 510
	Python 2	32 537
	C#	1 318
Big sorting	Java	2 116
	Python 2	1 518
	C#	51 755
Simple Array Sum	Java	108 916
	Python 2	40 054

A forráskódok összegyűjtése után az adatok feldolgozásra és átalakításra kerültek a következő lépések által:

- előfeldolgozás,
- tokenizálás,
- vektortér létrehozása.

3.2.3.1.1 Előfeldolgozás

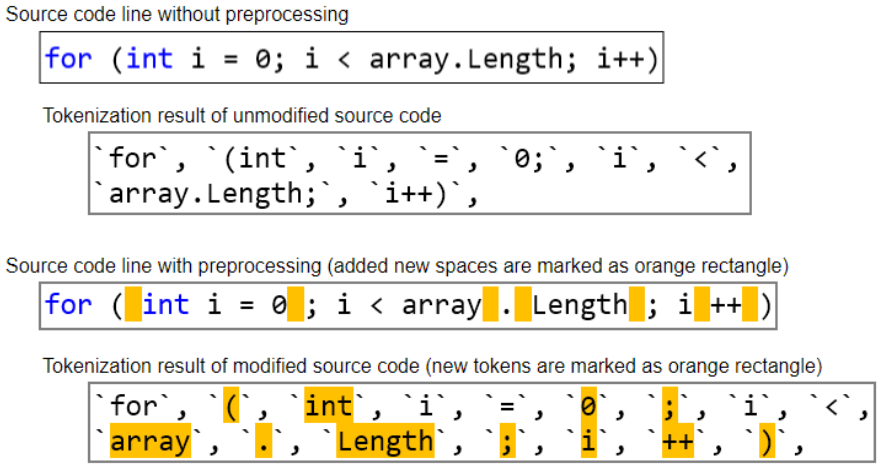
A vektorizálás előtt az adathalmaz előfeldolgozását kell elvégezni, melynek célja az adatok megtisztítása a nem kívánt elemektől. Ilyen elemek lehetnek megjegyzések, felesleges (duplikált) szóközök, tabulátorok, sorkezdő karakterek és bármely olyan rész, amely nem befolyásolja a program végrehajtását, figyelembe véve a programozási nyelv jellemzőit és sajátosságait. Például Pythonban a sor elején lévő szóközöknek (vagy tabulátoroknak) más a jelentésük, mint a soron belüli szóközöknek.

3.2.3.1.2 Tokenizálás

A természetes nyelvi feldolgozásban a tokenizáció az a mechanizmus, amely egy szöveget szavakra bont, majd ezeket a szavakat, mint tokeneket használja fel a további lépésekben. Forráskód esetén a tokenizáció ugyanígy érhető el, azzal a különbséggel, hogy különös figyelmet kell fordítani az

egyes karakterek jelentésére. [82,83]

A karakterek jelentését figyelembe véve a kódot újra kell formázni: szóközöket kell beszúrni az elsődleges parancsok elé és mögé, valamint speciális karaktereket, például zárójeleket és pontosvesszőket (ahol szükséges). Ily módon a tokenizáció helyesen hozza létre a tokeneket. Ez a módosítás nem befolyásolja a kód működését. E mechanizmus nélkül a tokenizáció eredménye hatalmas mennyiségű nem tanulható tokent tartalmazna, és az osztályozás sikertelen lenne. A **3.2.2.** ábra a tokenizáció eredményét mutatja ezzel a folyamattal és anélkül.



3.2.1. ábra. *Forráskódrészlet tokenizálás közötti különbség előfeldolgozással és anélkül.*

A következő (Pythonban írt) reguláris kifejezést használtam a tokenek lekéréséhez:

$$[!\backslash"'\#\$\%\& * +, -./:;<=>?@()[\backslash]\{\}|\hat{}\_\sim\backslash\backslash]|[\backslash w] +.$$

3.2.3.1.3 Vektortér létrehozása

Annak érdekében, hogy a hálózat megértse a tokenizált fájlokat, létre kell hozni a vektorteret, [84] amely képes leképezni a szavak mögöttes jelentését. Egy gondolattérképet (technikailag egy vektorteret), amely tartalmazza a token kapcsolatokat és azok mögöttes jelentését. A vektortérrel kapcsolatos fontos elvárás, hogy az összes releváns és előforduló tokent tartalmazza, mivel a forrásfájlok nem dolgozhatók fel olyan vektortérrel, amely nem képes minden tokent leképezni. A vektortérnek rendelkeznie kell azzal a tulajdonsággal is, hogy egyedi azonosítót rendeljenek a bennük lévő tokenekhez, például két különböző

parancsnak (egy nyelven belül) nem lehet ugyanaz az azonosítója.

A vektorterek létrehozásának számos módja létezik. [85] Ezen módszerek közül a CBOW módszer vált használhatóvá rövid szövegek esetén, mivel kisebb korpuszokból jobb vektorteret képes létrehozni. Nagyobb korpuszok esetén a Skip-Gram modell használata előnyösebb lehet. A vektortér a word2vec [86] segítségével lett legenerálva, ahol a végső vektortér vektorai 1×100 dimenziójúnak lettek választva. Így minden szót 100 szám reprezentál egy sorvektorban. [87,88]

A konvolúciós neurális hálózat bemenete egy mátrix. Ez a mátrix a vektortérből származó vektorok alapján kerül összeállításra. A mátrix *i*-edik sora megegyezik a forrásfájl *i*-edik tokenjével. A megoldások változó hossza és a CNN fix bemeneti mérete miatt a mátrix sorainak számát 200-nak választottam, mivel nem volt olyan forrásfájl, amely 200-nál több tokent tartalmazott volna. Az értékes tokeneket nem tartalmazó sorokat a nulla sorvektor reprezentálta. Ezek a mátrixok nem kerültek eltárolásra, mivel a létrehozásához szükséges összes információ a tárolt forrásfájlokból és a vektortérből származott.

3.2.3.1.4 Az adattisztítás és a vektortér minőségének ellenőrzése

A sikeres osztályozás feltétele a megfelelő minőségű adattisztítás, tokenizálás és az ezekből létrehozott vektortér. Ennek ellenőrzésére kettő vektorteret készítettem el, minden esetben azonos paraméterezéssel az eredmények összehasonlíthatóságának érdekében. Első esetben a nyers adatokból, második esetben a tisztítást követő adatokból készült el a vektortér. A nem feldolgozott esetben a vektortér építéséhez használt adathalmaz egyedi szavainak száma 509 519, mely 353 782 olyan szót tartalmazott, ami csak egyszer szerepel, míg a megtisztított adatokból létrehozott vektortér esetében 9 562 szóra csökkent az egyedi szavak száma, melyekből csak 3 316 olyan szó volt, ami egyszer fordult elő a megtisztított adathalmazban.

Egy létrehozott vektortér minőségét kvantitatív módon nehéz értékelni, ezért itt csak kvalitatív módszerek jöhetnek szóba. Mivel egy jó szóvektor tértől elvárható, hogy a hasonló jelentésű szavak egymáshoz hasonló vektorokra legyenek leképezve, ezért egy általánosan elfogadott vizsgálati módszer, hogy szúrópróbaszerűen ellenőrizzük néhány ismert, egymáshoz hasonló

jelentésű vektor különbségét.

Az elkészített vektorterek minőségét az alapján vizsgáltam, hogy egy adott szóhoz milyen szavak találhatóak közel a térben. Mindkét esetben vettem a programozási nyelv összes kulcsszavát, majd vizsgáltam a környezetükben található legközelebbi 5 szót. Elvárásom az volt, hogy a környezetben az azonos jelentéssel bíró kulcsszavak jelenjenek meg (például, ha a `for` volt a vizsgált szó, akkor annak környezetében vártam a többi ciklus definiálására alkalmas utasítást: `foreach`, `do`, `while`). A vektorterek a decimal, mint lebegőpontos számábrázolási típus kulcsszavára az alábbi hasonló szavakat adták:

- A vizsgált szó: *decimal*
- Elvárt szavak (lebegőpontos számábrázolási típusok): *float, Float, double, Double, Decimal*
- A legközelebbi öt szó a nyers adatokból létrehozott vektortérben:

```
0.00m
0m
Math.Round(decPosShare/intArrSize,6);
Decimal.Round(Convert.ToDecimal(n)/Convert.ToDecim
(decimal)pos/n;
```

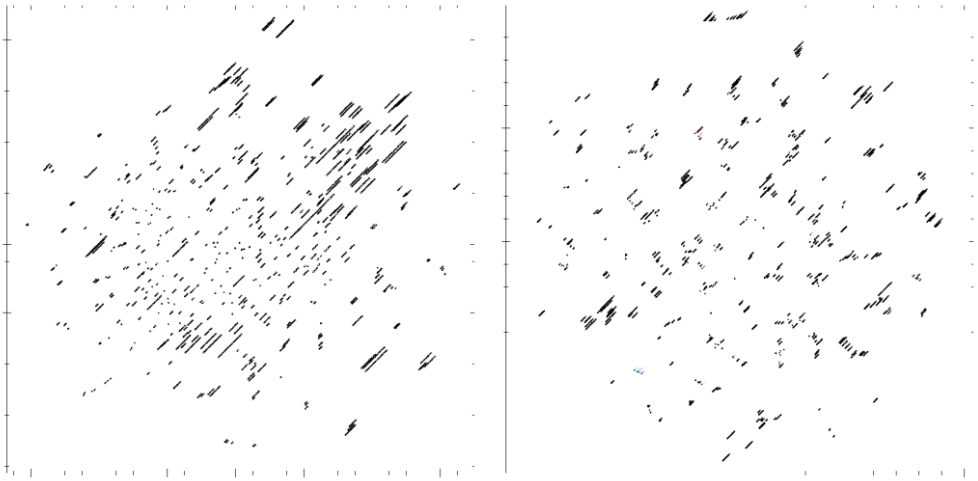
- A legközelebbi öt szó a tisztított adatokból létrehozott vektortérben:

```
Decimal
float
double
Double
0d
```

Összességében azon esetek száma, melyeknél legalább 3 helyes szó szerepelt az 5 legközelebbi között 43% volt a feldolgozatlan adatok vektorterében, és 79% volt a megtisztított adatokból előállított vektortér esetén, ami alapján megállapítható, hogy az adattisztítást követően az előállított vektortér minősége jelentősen javult. További javulást a vektortér előállításának paraméterezése, illetve a felhasznált adathalmaz méretének növelése jelentheti.

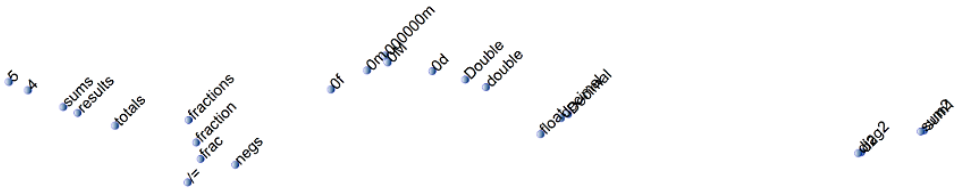
Végezetül az előállított vektorterek első 500 eleméből a TSNE (t-Distributed Stochastic Neighbor Embedding) algoritmus [89] segítségével készítettem egy kétdimenziós képet (**3.2.2.** ábra). A **3.2.2.** ábra (**A**) és (**B**)

képe között jól látható különbség, hogy a megtisztított adatokat felhasználva elkezdtek megjelenni az azonos jelentéssel bíró csoportok a térben (**B**), míg enélkül egy egyenletes eloszlású teret kaptam (**A**). A **3.2.2.** ábra (**C**) képe a „decimal” kulcsszó környezetét mutatja be.



*(**A**) A nyers adatok alapján. A szavak jól láthatóan nem csoportosulnak.*

*(**B**) A megtisztított adatok alapján. A szavak jól láthatóan csoportokat alkotnak.*



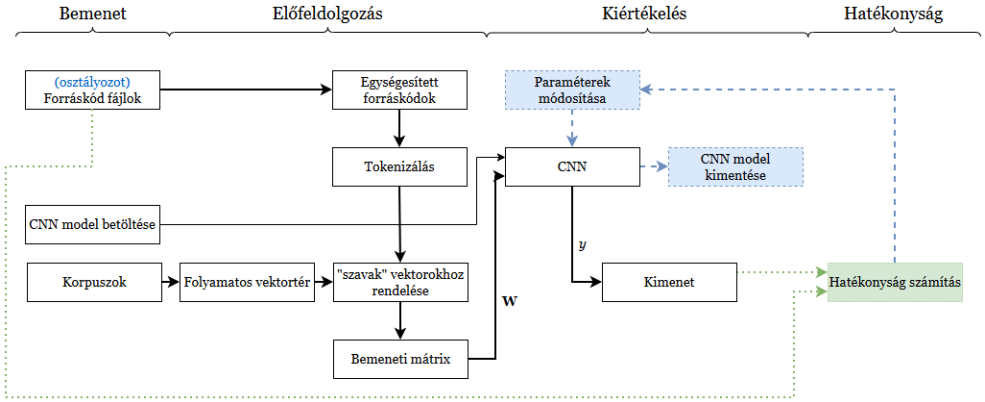
*(**C**) A megtisztított adatok alapján készült vektortér ábrázolásának, a példánál szereplő „decimal” környezetének kinagyítása.*

3.2.2. ábra. *A nyers, illetve megtisztított adatok első 500 eleme alapján készített vektorterek vizualizációja.*

3.2.3.2 A rendszer modellje

A teljes modell 4 különálló részből áll, ahol a Konvolúciós Neurális Hálózat a központi feldolgozó elem. Ezt megelőzi a bemeneti rész, amely a forráskódfájlok (a betanítás során a kimeneti osztállyal jelölve) és a korpuszok gyűjtésével (és ha van ilyen, az előre betanított CNN betöltésével) kezdődik. Ezután következik az előfeldolgozás, amely többek között az adatok tisztítását és tokenizálását foglalja magában, ami szorosan kapcsolódik a hálózat vektorterének és bemeneti mátrixainak létrehozásához. Az előfeldolgozás feladata a vektortér előállítása a

korpuszból. A modell működését egy kiértékelő logika zárja le, amelynek célja a betanítás minőségének és a modell helyes működésének ellenőrzése. A **3.2.3.** ábra a modell felépítését mutatja, ahol a kék szaggatott vonalak a betanításban használt elemeket jelzik.

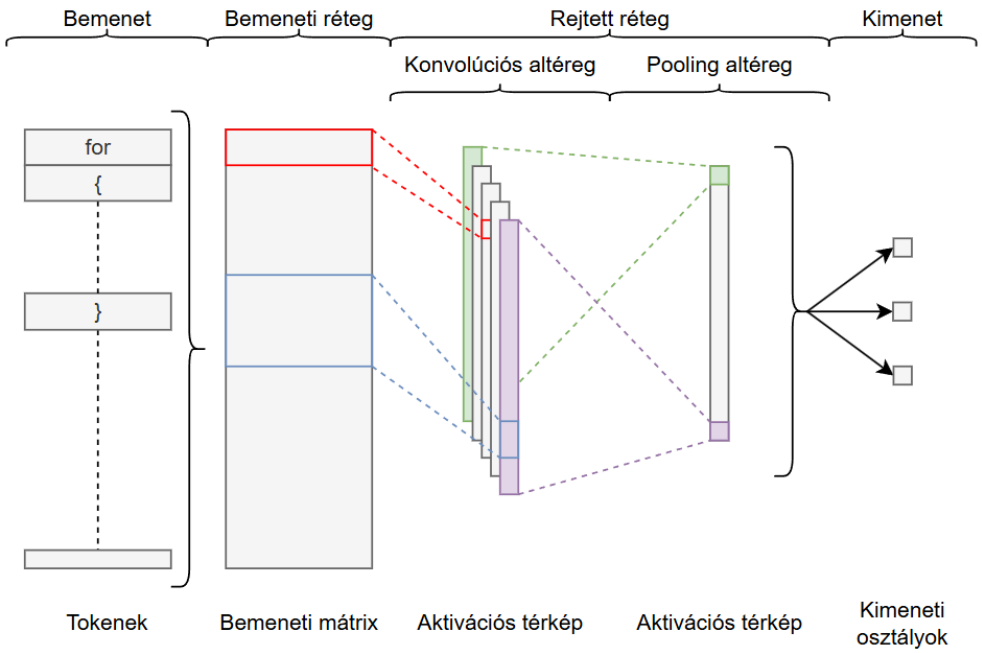


3.2.3. ábra. *A rendszer modelljének felépítése.*

A modell működése során a hálózatnak a tokenek sorozatából meg kell értenie az algoritmus felépítését és működését, vagyis a hálózatnak képesnek kell lennie megkülönböztetni az utasításokat és azok sorrendjét. Fontos megjegyezni azt is, hogy a hálózat nem felelős a kód helyességének kiértékeléséért (a kód nem kerül végrehajtásra), mivel azt a HackerRank egységtesztjei garantálják.

3.2.3.2.1 CNN a forráskódfeldolgozásban

A CNN egy sor konvolúciós rétegen alapul, amelyek célja a releváns információk kiemelése a bemenetből (jelen esetben a forráskódokból, ahol elegendő egyetlen konvolúciós réteg). Ezeket a rétegeket követi a pooling réteg, amely a kiemelt információkat egyetlen vektorrá összesíti. Végül az összesített értéket a teljesen összekapcsolt réteg osztályozza. A CNN architektúrája a **3.2.4.** ábrán látható.



3.2.4. ábra. *CNN architektúrája forráskód feldolgozás során.*

A hálózat bemenete a tokenizált forrásfájlon és a vektortéren alapuló mátrix (csak azok a forrásfájlok alakulnak át bemeneti mátrixokká, ahol minden tokennek van egy vektora a vektortérben). A CNN első rétege a bemeneti réteg, amelynek egyetlen feladata a bemeneti mátrixok fogadása és továbbítása az első rejtett réteg, a konvolúciós réteg felé.

A bemeneti réteget az első rejtett réteg követi, amely a konvolúciós műveletet végrehajtja a magas szintű információk kinyerésére a bemeneti mátrixból a **3.1.1** képlet szerint. A réteg 1-től 6-ig terjedő szűrőméretet használ, ami öt vektort eredményez. Ezt az hat vektort a következő réteg, a pooling réteg dolgozza fel a ReLu nemlineáris aktivációs függvénnyel, mivel ez gyors és pontos betanítását biztosítja a hálózatnak. [90] A pooling réteg a **3.1.2** és a **3.1.3** képlet szerinti Max-Pooling módszert alkalmazza. Ez a művelet nem hoz új paramétert a hálózatba, bár hátránya, hogy a hálózat könnyen túltanulhatja magát.

Végül a hálózat egy teljesen összekapcsolt réteggel zárul, egy egyszerű, előrecsatolt neurális hálózattal SoftMax aktivációs függvényekkel a **3.1.4** képlet szerint. Ez a réteg valószínűségeloszlás alapján osztályozza az aggregált információkat.

Minden forrásfájl kimenete egy 3 elemű vektor lesz, amely megmutatja,

hogy a bemenet mennyire tartozik az egyes kimeneti osztályokba. Ezen valószínűség alapján kell eldönteni, hogy a forrás melyik feladatot valósítja meg. Például a bemenet 32%-a „Very big sum” osztályba, 14%-a a „Simple array sum” osztályba, 54%-a pedig a „Big sorting” osztályba tartozik. Ebben az esetben a megoldást a „Big sorting” megoldásának tekintjük.

A betanítás során a forráskódok kiértékelésének eredményeitől függően módosulnak a hálózati paraméterek. Ezek a paraméterek a következők: a bemeneti mátrix mérete, a konvolúciós réteg szűrőinek mérete és a pooling típusa.

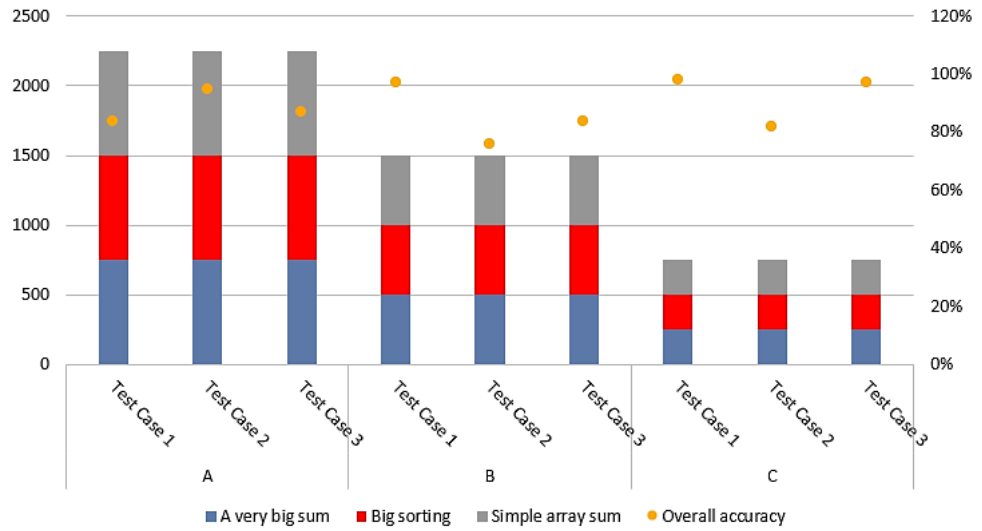
3.2.3.2.2 Osztályozási és betanítási adatok kiválasztása

A feldolgozás két fázisra osztható a kívánt funkcionalitástól (tanítás vagy előhívás) függően. A tanítási fázisban a rendelkezésre álló forrásfájlokat két halmazra kell osztani, tanító és validáló halmazokra. Ha rendelkezésre áll egy előre betanított hálózat, akkor elegendő a hálózatot betölteni a modellbe, és átadni annak az adatokat a kimenet előállításához.

A modell működésének ellenőrzéséhez egy százalékos érték (pontosság) került meghatározásra, amely a helyesen kiértékelt adatok száma osztva a validáció során felhasznált adathalmazok számával.

A modellt először Java nyelvű megoldásokkal teszteltem. Az eredményeket a **3.2.5.** ábra mutatja. Az első esetben (**3.2.5.** (**A**)) a hálózat egy nagyobb tanítóhalmazzal lett tanítva (minden feladatból 750), ami a teljes halmaz 89%-os pontosságát eredményezte a validáció után. Ezt követte a tanítóadatok számának 500-ra (**3.2.5.** (**B**)), majd 250-re (**3.2.5.** (**C**)) csökkentése. A futásokból (tanítóhalmaz méret szerint) kiválasztottam az elsőt, amiknek igazságmátrixait a **3.2.2.** táblázatban látható, illetve az ezekből számított értékeket a **3.2.3.** táblázat tartalmazza.

Az igazságmátrixok (**3.2.2.** táblázat) és az azokból számított értékek (**3.2.3.** táblázat) alapján megállapítható, hogy a modell megfelelően robosztus volt, a „Big sorting” típusú feladatok osztályozása konzisztensen magas, míg a másik két feladatot gyakrabban tévesztette össze azok jellege miatt.



3.2.5. ábra. *Java feladatok fix tanulóhalmaz-választással és pontossággal, feladatonként (A, B, C) háromszor futtatva azonos környezetben és paraméterezéssel („Test Case”). Az „A” esetben a feladatokból 750 darab, a „B” esetében 500 darab és a „C” esetében 250 darab elem jelentette a tanító halmaz elemeinek a számát feladatonként.*

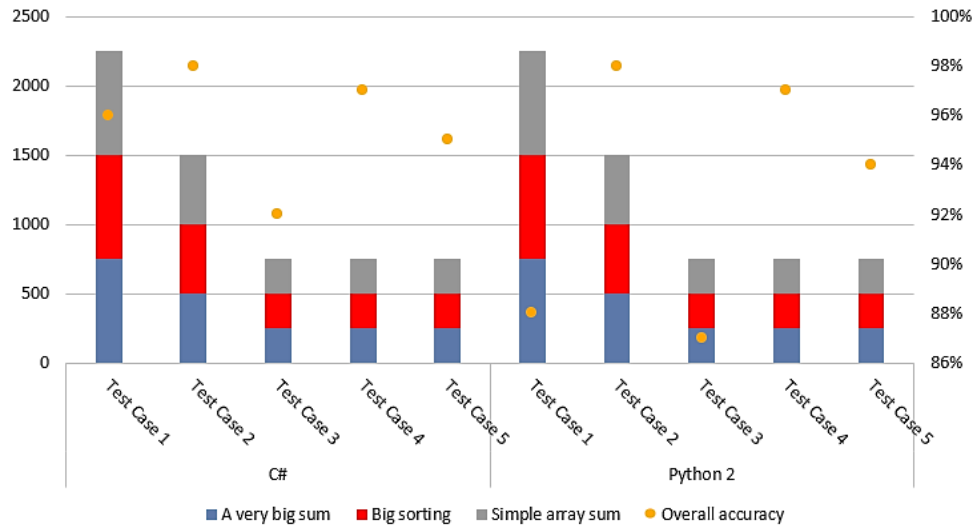
3.2.2. táblázat. *A Java nyelvű feladatmegoldások validációs eredményeinek igazságmátrixai.*

Java 750 – Test Case 1		Prediktált					
		A very big sum		Big sorting		Simple array sum	
Tényleges	A very big sum	VP_{AA}	**563**	HN_{AB}	**17**	HN_{AC}	**170**
	Big sorting	HN_{BA}	**13**	VP_{BB}	**718**	HN_{BC}	**19**
	Simple array sum	HN_{CA}	**118**	HN_{CB}	**23**	VP_{CC}	**609**
Java 500 – Test Case 1		Prediktált					
		A very big sum		Big sorting		Simple array sum	
Tényleges	A very big sum	VP_{AA}	**477**	HN_{AB}	**5**	HN_{AC}	**14**
	Big sorting	HN_{BA}	**3**	VP_{BB}	**495**	HN_{BC}	**4**
	Simple array sum	HN_{CA}	**12**	HN_{CB}	**7**	VP_{CC}	**483**
Java 250 – Test Case 1		Prediktált					
		A very big sum		Big sorting		Simple array sum	
Tényleges	A very big sum	VP_{AA}	**243**	HN_{AB}	**2**	HN_{AC}	**5**
	Big sorting	HN_{BA}	**0**	VP_{BB}	**247**	HN_{BC}	**2**
	Simple array sum	HN_{CA}	**5**	HN_{CB}	**1**	VP_{CC}	**245**

3.2.3. táblázat. *A* ***3.2.2.*** *táblázat igazságmátrixaiból számított értékek.*

	Java 750 Test Case 1	**Java 500 Test Case 1**	**Java 250 Test Case 1**
Valódi pozitív arány			
$VPR_A = \frac{VP_{AA}}{VP_{AA} + HN_{AB} + HN_{AC}}$	0,750	0,961	0,972
$VPR_B = \frac{VP_{BB}}{VP_{BB} + HN_{AB} + HN_{AC}}$	0,957	0,986	0,989
$VPR_C == \frac{VP_{CC}}{VP_{CC} + HN_{AB} + HN_{AC}}$	0,812	0,962	0,974
Pozitív prediktív érték			
$\mathrm{PPE}_A = \frac{VP_{AA}}{VP_{AA} + HN_{BA} + HN_{CA}}$	0,811	0,969	0,979
$\mathrm{PPE}_B = \frac{VP_{BB}}{VP_{BB} + HN_{AB} + HN_{CB}}$	0,947	0,976	0,988
$\mathrm{PPE}_C = \frac{VP_{CC}}{VP_{CC} + HN_{AC} + HN_{BC}}$	0,763	0,964	0,972
Pontosság			
$\frac{VP_{AA} + VP_{BB} + VP_{CC}}{\text{Összes elem}}$	0,840	0,970	0,980
F1 pontszám			
$F1_A = \frac{2 * PPE_A * VPR_A}{PPE_A + VPR_A}$	0,779	0,965	0,975
$F1_B = \frac{2 * PPE_B * VPR_B}{PPE_B + VPR_B}$	0,952	0,981	0,989
$F1_C = \frac{2 * PPE_C * VPR_C}{PPE_C + VPR_C}$	0,786	0,963	0,974
Kritikus sikerességi mutató			
$KSI_A = \frac{VP_{AA}}{VP_{AA} + HN_{AB} + HN_{AC} + HN_{BA} + HN_{CA}}$	0,639	0,933	0,952
$KSI_B = \frac{VP_{BB}}{VP_{BB} + HN_{BA} + HN_{BC} + HN_{AB} + HN_{CB}}$	0,908	0,963	0,980
$KSI_C = \frac{VP_{CC}}{VP_{CC} + HN_{CA} + HN_{CB} + HN_{AC} + HN_{BC}}$	0,648	0,925	0,949

Az előző, bíztató eredmények alapján a modellt C# és Python 2 nyelven is teszteltem, melynek eredményeit a **3.2.6.** ábra szemlélteti, az igazságmátrixok és az azokból számított értékeket a **3.2.4.** táblázat és a **3.2.5.** táblázat mutatja be. A futásokból mindkét feladat esetében kiválasztottam az utolsót, ahol a tanító halmaz mérete 250/feladattípus volt. A számított értékek (**3.2.5.** táblázat) alapján elmondható, hogy C# nyelv esetében kiegyensúlyozottan magas értéket mutat mindhárom kategóriában, melyek közül a Java megoldásokhoz hasonlóan a „Big sorting” feladat megoldásait sikerült a leginkább helyesen osztályozni, a tévesztések száma elhanyagolható. Sajnos azonban a hasonló feladatok esetében keresztirányú tévesztés enyhén visszaveti az osztályok kritikus sikerességi mutatóit. Python 2-ben megoldott feladatok esetén a pontosság minimálisan alacsonyabb, mint a C# esetén, illetve itt is megfigyelhető a két hasonló feladat („A very big sum” és a „Simple array sum”) közötti magasabb tévesztés.



3.2.6. ábra. *C# és Python 2 nyelven készített feladatok tanítása és pontossága, nyelvenként ötször futtatva azonos környezetben és paraméterezéssel, de különböző adathalmaz mérettel („Test Case”). A „Test Case 1” esetében mindhárom feladattípusból 750 darab, a „Test Case 2” esetében 500, a „Test Case 3, 4, 5” esetében pedig 250 darab került kiválasztásra véletlenszerűen.*

3.2.4. táblázat. *A C# és Python2 nyelvű feladatmegoldások validációs eredményeinek igazságmátrixai.*

C# – Test Case 5		Prediktált					
		A very big sum		Big sorting		Simple array sum	
Tényleges	A very big sum	VP_{AA}	**237**	HN_{AB}	**1**	HN_{AC}	**15**
	Big sorting	HN_{BA}	**2**	VP_{BB}	**241**	HN_{BC}	**5**
	Simple array sum	HN_{CA}	**11**	HN_{CB}	**2**	VP_{CC}	**236**
Python2 – Test Case 5		Prediktált					
		A very big sum		Big sorting		Simple array sum	
Tényleges	A very big sum	VP_{AA}	**228**	HN_{AB}	**6**	HN_{AC}	**15**
	Big sorting	HN_{BA}	**3**	VP_{BB}	**295**	HN_{BC}	**3**
	Simple array sum	HN_{CA}	**12**	HN_{CB}	**6**	VP_{CC}	**182**

3.2.5. táblázat. *A* ***3.2.4.*** *táblázat igazságmátrixaiból számított értékek.*

	C# 250 Test Case 5	**Python2 250 Test Case 1**
Valódi pozitív arány		
$VPR_A = \frac{VP_{AA}}{VP_{AA} + HN_{AB} + HN_{AC}}$	0,936	0,915
$VPR_B = \frac{VP_{BB}}{VP_{BB} + HN_{AB} + HN_{AC}}$	0,971	0,983
$VPR_C == \frac{VP_{CC}}{VP_{CC} + HN_{AB} + HN_{AC}}$	0,947	0,905
Pozitív prediktív érték		
$\mathrm{PPE}_A = \frac{VP_{AA}}{VP_{AA} + HN_{BA} + HN_{CA}}$	0,948	0,938
$\mathrm{PPE}_B = \frac{VP_{BB}}{VP_{BB} + HN_{AB} + HN_{CB}}$	0,987	0,960
$\mathrm{PPE}_C = \frac{VP_{CC}}{VP_{CC} + HN_{AC} + HN_{BC}}$	0,921	0,910
Pontosság		
$\frac{VP_{AA} + VP_{BB} + VP_{CC}}{\text{Összes elem}}$	0,952	0,940
F1 pontszám		
$F1_A = \frac{2 * PPE_A * VPR_A}{PPE_A + VPR_A}$	0,942	0,926
$F1_B = \frac{2 * PPE_B * VPR_B}{PPE_B + VPR_B}$	0,979	0,971
$F1_C = \frac{2 * PPE_C * VPR_C}{PPE_C + VPR_C}$	0,934	0,907
Kritikus sikerességi mutató		
$KSI_A = \frac{VP_{AA}}{VP_{AA} + HN_{AB} + HN_{AC} + HN_{BA} + HN_{CA}}$	0,890	0,863
$KSI_B = \frac{VP_{BB}}{VP_{BB} + HN_{BA} + HN_{BC} + HN_{AB} + HN_{CB}}$	0,960	0,945
$KSI_C = \frac{VP_{CC}}{VP_{CC} + HN_{CA} + HN_{CB} + HN_{AC} + HN_{BC}}$	0,877	0,831

3.2.4 A bemutatott módszer értékelése

A célként kitűzött programozási feladatok osztályozását sikerült megoldani egy CNN segítségével, ahol az osztályozás kumulatív pontossága végül meghaladta a 95%-ot. A modellt 250 (Java / C# / Python 2) forráskóddal tanítottam, és 197 542 / 93 490 / 74 109 kóddal teszteltem, a pontosság 95% / 96% / 93% volt. Ezen eredmények eléréséhez fontos szerepet játszott a jó minőségű adatok kiválasztása, amit garantált az egységteszteken való megfelelés. Az osztályozási pontosság eredményei között nem volt szignifikáns különbség, azaz a használt forráskód nyelvének megválasztása nem befolyásolta a betanított hálózat osztályozási képességét.

3.2.5 Konklúzió

Az eredmények ígéretesek és további kutatásokat igényelnek, elsősorban a feladatok számának növelésében (három feladat helyett több tucat feladat). A továbbfejlesztés egy másik lehetősége a modell osztályozása, amennyiben az osztályozás képességét leválasztják a programozási nyelvről.

Az eredmények alapján a modell elsődleges felhasználhatósági lehetősége az oktatás területén rejlik, ahol a hallgatókkal végzett programozási vizsgák feladatai a megoldott feladatokhoz rendelhetők. Alternatív megoldásként az eredmények jelentősek lehetnek az iparban is. Ha a modell képes lenne kódrészleteket osztályozni a forrásfájlok helyett, így elkerülve a kódduplikációt, jelentős időmegtakarítás lenne elérhető. A kódduplikáció megakadályozása olyan rendszerekben is hasznos lehet, ahol korlátozott a memória vagy a tárhely, például a beágyazott rendszerekben. Továbbá a nem duplikált kód elősegíti a rendszer tervezési problémáinak (például az öröklődés hiányának) javítását, illetve a karbantarthatóságot.

3.2.6 Tézis kimondása

2.2. tézis – Kidolgoztam egy újszerű módszert, amely alkalmas forráskódok előre megadott osztályokba sorolására. Megoldásom a természetes nyelvi feldolgozáshoz kidolgozott szóvektor modellt adaptálja forráskód tokenekre, továbbá bevezet egy erre a célra specializált adattisztítási eljárást. Az így felépített szóvektor tér és egy konvolúciós neurális hálózat segítségével az általam kidolgozott modell képes forráskódokhoz valószínűségeket rendelni

aszerint, hogy azok a tanítási adatbázis melyik osztályába illeszkednek. A valós programokon futtatott validáció igazolja, hogy a módszer nagy pontossággal képes a forráskódokról megállapítani, hogy azok melyik feladatot oldják meg. [S5, S6, S7]

3.3 Eredmények értékelése

A **3.1** fejezet a rövid szöveges válaszok automatikus kiértékelését mutatja be konvolúciós neurális hálózaton alapuló modell segítségével. A bemutatott rendszer képes a válaszokat tartalmuk alapján kiértékelni, elkerülve a kulcsszavas megközelítések tipikus problémáját, a kontextus helyes értelmezését. A rendszer különösen hatékony volt az angol nyelvű, strukturáltabb válaszok esetében, amit a közel 85–87%-os osztályozási pontosság is alátámasztott. A magyar nyelvű hallgatói válaszokkal végzett tesztek során azonban nehézségek adódtak a helyes kiértékelésnél, amit az adathalmaz mérete és minősége indokolt. Az eredmények alapján a modell teljesítménye szignifikánsan javult nagyobb tanítóhalmaz esetén, míg kisebb adatkészletek esetén romlott az osztályozás megbízhatósága. Fontos megjegyezni, hogy a természetes nyelvfeldolgozás (NLP) területén a modell teljesítménye közvetlen összefüggésben áll a vektortér minőségével és a tanítóadatok sokszínűségével.

A **3.2** fejezetben tárgyalt módszer a szóvektor-alapú tokenizálásra és CNN-re épülő megközelítés forráskódok osztályozására egy újfajta módszer volt a gépi tanulás kódosztályozás alkalmazására. A módszer különlegessége, hogy nyelvfüggetlen tokenizálást tesz lehetővé, ezáltal képes különböző programozási nyelven írt megoldásokat is azonos vektortérben kezelni. A teszteredmények azt mutatták, hogy a modell három különböző programozási nyelv (Java, C#, Python 2) esetén is 93–96%-os osztályozási pontosságot ért el. Az előfeldolgozás, különösen a tokenizáció helyes kialakítása, kulcsszerepet játszott abban, hogy a CNN megfelelően tanulhassa meg az algoritmusok szerkezeti és szintaktikai sajátosságait. A modell korlátját ugyanakkor az jelenti, hogy a CNN fix bemeneti méretet igényel, amelyet különböző technikákkal kell biztosítani – ez bizonyos esetekben információvesztéssel járhat.

3.3.1 Továbbfejlesztési lehetőségek, jövőbeni kutatási irányok

A bemutatott módszerek, bár sikeresnek bizonyultak az adott feladatokban, számos ponton továbbfejleszthetők, illetve kiterjeszthetők más alkalmazási területekre is.

3.3.1.1 A rövid szöveges válaszok kontextusérzékeny értékelésének kialakítása

A jelenlegi megközelítés elsősorban a kulcsszavak jelenlétére és azok kombinációira épül. Egy következő szint lenne, ha a modell képes lenne a válaszok szemantikai elemzésére és a kontextuális kapcsolatok értelmezésére is. Itt olyan nyelvi modellek alkalmazása jöhet szóba, amelyek figyelemmechanizmusokat alkalmaznak (pl. Transformer architektúrák), és képesek a mondat szintű reprezentációk összehasonlítására. Ennek révén a modell a tartalom mélyebb megértésére lenne képes.

3.3.1.2 Transfer learning alkalmazása forráskódok osztályozásánál

A forráskódok osztályozására bemutatott CNN-modell statikusan tanul egy konkrét feladathalmaz alapján. A modell széleskörű használhatósága érdekében célszerű lenne a transfer learning alkalmazása, amely lehetővé tenné, hogy a már betanított modellt új feladatokra alkalmazzuk, minimális újratanítással. Ez különösen hasznos lehet kisebb adathalmazok esetén, vagy olyan esetekben, amikor új típusú algoritmusokat kell besorolni a már ismert osztályok mellé.

3.3.1.3 Kódszintű granularitás növelése

A jelenlegi forráskódokat osztályozó modell teljes forráskódfájlokat dolgoz fel, azonban a gyakorlati alkalmazások szempontjából hasznos lenne a függvény- vagy kódrészlet-alapú kiértékelés is. Ennek megvalósításához új tokenizálási és előfeldolgozási stratégiák kidolgozására lenne szükség, amelyek képesek a forráskód funkcionális részeinek elkülönítésére és feldolgozására. Ez előnyös lenne például kódklónok keresésében vagy refaktorálási lehetőségek automatikus felderítésében.

Összegzésül elmondható, hogy a 3. fejezetben bemutatott gépi tanulási modellek megalapozzák a rövid szöveges válaszok és forráskódok automatikus értékelésének hatékony módszertanát. Az eddigi eredmények biztatóak, de a gyakorlati alkalmazásukhoz szükség van a rendszerek rugalmasságának és általánosíthatóságának javítására. A jövőbeni kutatásoknak ezt a kettő irányt kell fókuszba helyezniük, hogy a módszerek szélesebb körben és nagyobb megbízhatósággal legyenek alkalmazhatók.

4 Összefoglalás

A kutatásom során kidolgoztam nem gépi tanuláson alapuló forráskód-összehasonlító módszert, ami a leghosszabb közös részstring (LCS) elvét iteratív módon alkalmazza, amely a hagyományos szövegalapú metrikáknál hatékonyabban kezeli a programkódok szerkezeti sajátosságait, például a független utasítások felcserélhetőségét. A megoldásom nagy előnye, hogy programozási nyelvtől független, és nem igényel komplex szintaktikai előfeldolgozást. A nagy számítási igényből adódó hátrányokat egy újszerű, GPU-alapú adatpárhuzamosítással , valamint az egymásba ágyazott ciklusok reflexív, szimmetrikus, nem tranzitív relációira optimalizált, terheléskiegyenlített párhuzamosításával küszöböltem ki. A particionálási stratégia 10 000 elem feletti adathalmazoknál 15–40%-os futásidő-csökkenést eredményezett. Gyakorlati szempontból ezek a módszerek kiválóan alkalmazhatók nagy méretű kódbázisok karbantartása során duplikátumok azonosítására vagy plágiumellenőrzésére, ahol a gyors és skálázható feldolgozás elengedhetetlen.

A hagyományos módszerek jövőbeni továbbfejlesztése több irányban is lehetséges. Az LCS-alapú összehasonlítás pontossága és megbízhatósága növelhető egy olyan kombinált megközelítéssel, amely a karakter és token szintű hasonlóságot egyaránt figyelembe veszi, így kiszűrve a false pozitív egyezéseket. A GPU-s gyorsítás skálázhatósága terén célszerű lenne az adatmozgatás és a kernelindítás átfedésének megvalósítása, illetve több szöveg egyidejű összehasonlítása, ami tovább csökkentené a memóriaátviteli késleltetéseket. Az indexalapú particionálásnál pedig egy bemenet vezérelt, dinamikus algoritmus kidolgozása jelentené a következő lépést, amely az adathalmaz mérete, szerkezete és redundanciája alapján automatikusan választaná meg a optimális párhuzamosítási stratégiát.

A gépi tanuláson alapuló kutatásaim során két fő területet vizsgáltam, a rövid szöveges válaszok automatikus kiértékelését és a forráskódok osztályozását. A konvolúciós neurális hálózatokra (CNN) épülő szövegértékelő modell képes volt túllépni az egyszerű kulcsszókeresés korlátain azáltal, hogy a válaszok tartalmát és kontextusát is értelmezte. Míg a megfelelő méretű, angol nyelvű adathalmazokon a modell magas, 85–87%-os pontossággal teljesített, a kisebb magyar nyelvű tanulóhalmazokon

végzett tesztek rávilágítottak a módszer erős adatigényére és a vektortér minőségének fontosságára. A forráskódok osztályozására kidolgozott, szóvektor alapú tokenizálásra épülő CNN modell ezzel szemben rendkívül robusztusnak bizonyult. Nyelvfüggetlen tokenizálásának köszönhetően Java, C# és Python 2 forráskódok esetében is 93–96%-os pontosságot ért el. E modellek gyakorlati jelentősége kiemelkedő az oktatásban, de az iparban is hasznosíthatók a kódduplikációk elkerülésére.

A neurális hálózatokon alapuló modellek kiterjesztése során a kontextuális megértés elmélyítése a legfőbb jövőbeni feladat. A rövid szöveges válaszok értékelésénél a figyelemmechanizmusokat alkalmazó modern nyelvi modellek (például Transformer architektúrák) bevezetése jelentősen javíthatná a mondatszintű szemantikai elemzés minőségét. A forráskód-osztályozó hálózat általánosíthatósága a transzfertanulás integrálásával fokozható, ami lehetővé tenné a már betanított modell alkalmazását új feladatokra minimális újratanítás mellett. Emellett kiemelt kutatási irány a kódszintű felbontás növelése is, így a teljes forrásfájlok helyett a függvényszintű vagy kódrészlet alapú kiértékelés megvalósítása, amely elengedhetetlen a kódklónok precíz azonosításához és a refaktorálási lehetőségek automatikus felderítéséhez.

4.1 Tézisek

4.1.1 1. Téziscsoport – Nem gépi tanuláson alapuló összehasonlítási módszerek

1.1. tézis – Újfajta mérőszámot dolgoztam ki forráskódok szövegalapú összehasonlítására, amely a leghosszabb közös részstring (Longest Common Substring - LCS) keresés iteratív alkalmazásán alapul. Ezen mérőszám és egy megfelelően választott határérték segítségével egy új módszert dolgoztam ki, amelyik két forráskódról képes megállapítani, hogy azok ugyanazt a feladatot oldják-e meg. A módszert 4 különböző feladatot megoldó, 400 darab ember által készített forráskód egymással való összehasonlításával validáltam, amely alapján igazolni tudtam, hogy az jelentősen magasabb F1 score értéket ért el, mint a meglévő hasonló célú módszerek. [S1]

1.2. tézis – Kidolgoztam egy újszerű adatpárhuzamos módszert szövegek hasonlóságának mérésére. A módszer a leghosszabb közös részstring alapú összehasonlító módszer iteratív megvalósításán alapul, az abban található erőforrásigényes összehasonlításokat, illetve a megtalált részstringek eltávolítását azonban jóval hatékonyabban, adatpárhuzamos módon végzi el. A dinamikus programozás alapú szekvenciális és a GPU-n implementált adatpárhuzamos módszerek összehasonlítása alapján megállapítottam, hogy a nagyon rövid szövegek kivételével az általam kidolgozott módszer jelentősen kisebb futásidőt igényel, miközben azonos pontosságot ér el. [S2]

1.3. tézis – Kidolgoztam egy újszerű particionáliási eljárást, ami alkalmas egy halmaz elemeinek reflexív, szimmetrikus, nem tranzitív reláció alapú kompatibilitási osztályokba sorolásának hatékony párhuzamosítására. Az általam kidolgozott módszer a szükséges összehasonlító műveleteket a meglévő hasonló megoldásokhoz képest egyenletesebben osztja szét a megadott számú végrehajtó egység között. A módszer igazolásához számos tesztet futtattam, amelyek igazolták, hogy egy minimális elemszám felett az általam kidolgozott particionálás jelentősen kisebb futásidőhöz vezet, mint a meglévő hasonló felosztó módszerek. [S3]

4.1.2 2. Tézis csoport – Gépi tanuláson alapuló összehasonlítási módszerek

2.1. tézis – Kidolgoztam egy módszert, amely alkalmas szabadszavas rövid szöveges válaszok kiértékelésére. A módszer az eredetileg képfeldolgozási területen használt konvolúciós neurális hálózatokon alapul, kiegészítve azt speciális elő- és utófeldolgozó lépésekkel, amelyek jelentősen növelik a hálózat hatékonyságát a megadott területen. A valódi adatokon történő vizsgálatok kimutatták, hogy megfelelő méretű tanítási adatbázis megléte esetén az általam javasolt modell nagyobb pontosságot ért el, mint a hagyományos megoldások. [S4]

2.2. tézis – Kidolgoztam egy újszerű módszert, amely alkalmas forráskódok előre megadott osztályokba sorolására. Megoldásom a természetes nyelvi feldolgozáshoz kidolgozott szóvektor modellt adaptálja forráskód tokenekre, továbbá bevezet egy erre a célra specializált adattisztítási eljárást. Az így felépített szóvektor tér és egy konvolúciós neurális hálózat segítségével az általam kidolgozott modell képes forráskódokhoz valószínűségeket rendelni

aszerint, hogy azok a tanítási adatbázis melyik osztályába illeszkednek. A valós programokon futtatott validáció igazolja, hogy a módszer nagy pontossággal képes a forráskódokról megállapítani, hogy azok melyik feladatot oldják meg. [S5, S6, S7]

Saját publikációk

[S1] Á. Pintér and S. Szénási, **"Longest Common Subsequence-based Source Code Similarity"**, *2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Timisoara, Romania, 2023, pp. 000123-000128, doi: 10.1109/SACI58269.2023.10158551

Az [S1] publikáció független hivatkozásai:

1. Yoon, SeongCheol, Su-Hyun Kim, and Im-Yeong Lee. "A Study on PDG-based Source Code Similarity Checking Tools." Journal of Software Assessment and Valuation (한국소프트웨어감정평가학회 논문지) 19.4 (2023): 159-168.
2. Avitan, Matan, Elena V. Ravve, and Zeev Volkovich. "Assembly Function Recognition in Embedded Systems as an Optimization Problem." Mathematics 12.5 (2024): 658.

[S2] Á. Pintér and S. Szénási, **"GPU Acceleration of Longest Common Substrings Algorithm"**, *2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Timisoara, Romania, 2023, pp. 000189-000194, doi: 10.1109/SACI58269.2023.10158638

Az [S2] publikáció független hivatkozásai:

1. Gupta, Maahin, and Ayush Joshi. "Longest Common Substring in Pattern Matching." 2024 8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC). IEEE, 2024.

[S3] Á. Pintér and S. Szénási, **"Index Dependent Nested Loops Parallelization with an Even Distributed Number of Steps"**, *INFORMATICA-JOURNAL OF COMPUTING AND INFORMATICS 45*: 4 pp. 493-506., 16 p. (2022), doi: 10.31449/inf.v45i4.3130

[S4] Á. Pintér, B. Schmuck, and S. Szénási. **"Short text evaluation with neural network."** *Pollack Periodica* 13.3 (2018): 107-118. doi: 10.1556/606.2018.13.3.11

Az [S4] publikáció független hivatkozásai:

1. Saleh, Ali, and László Gáspár. "Optimizing asphalt foaming using neural network." Pollack Periodica 19.1 (2024): 130-136.

[S5] Á. Pintér and S. Szénási, "**Automatic Analysis and Evaluation of Student Source Codes,**" 2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2020, pp. 000161-000166, doi: 10.1109/CINTI51262.2020.9305819

Az [S7] publikáció független hivatkozásai:

1. Muepu, Daniel M., Yutaka Watanobe, and Md Faizul Ibne Amin. "A Comprehensive Content-Based Recommendation System for Programming Problems Through Multi-Faceted Code Analysis." *IEEE Access* (2025).
2. Horváth, Marek, Tomáš Kormaník, and Jaroslav Porubän. "Adaptation of Automated Assessment System for Large Programming Courses." *5th International Computer Programming Education Conference (ICPEC 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.
3. Angelovski, Damjan, Emil Stankov, and Mile Jovanov. "DEMAx Tool Based on an Improved Model for Semiautomatic C/C++ Source Code Assessment." *Proceedings of the 6th International Conference on Information and Education Innovations*. 2021.
4. Kalpana, R. A., S. Karunya, and R. Rashmi. "Student Academic Analyser and Career Guidance System Using Data Analytics and Visualization Techniques." *2023 Intelligent Computing and Control for Engineering and Business Systems (ICCEBS)*. IEEE, 2023.

[S6] Á. Pintér and S. Szénási, "**Comparison of Source Code Storage Methods,**" 2019 IEEE 19th International Symposium on Computational Intelligence and Informatics and 7th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Sciences and Robotics (CINTI-MACRo), Szeged, Hungary, 2019, pp. 000231-000236, doi: 10.1109/CINTI-MACRo49179.2019.9105260

[S7] Á. Pintér and S. Szénási, **"Classification of source code solutions based on the solved programming tasks,"** *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, Budapest, Hungary, 2018, pp. 000277-000282, doi: 10.1109/CINTI.2018.8928201.

Az [S6] publikáció független hivatkozásai:

1. Ma, Wei. "Next Generation Mutation Testing: Continuous, Predictive, and ML-enabled." (2022).
2. Ma, Wei, et al. "Graphcode2vec: Generic code embedding via lexical and program dependence analyses." *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022.
3. Liu, Chunhong, et al. "TLSCG: Transfer Learning-Based Efficient Anomalous Smart Contract Generation to Empower Unknown Vulnerability

Detection." *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 2024.

4. Ye, Weiwei, et al. "Detecting Unknown Threats in Smart Contracts With Domain Adaptation." *2024 IEEE International Conference on Software Services Engineering (SSE)*. IEEE, 2024.

Tézisekhez nem kapcsolódó saját publikációk

[S8] N. Neumann and Á. Pintér, "**Solving ARC with non-procedural program induction,**" 2023 IEEE 23rd International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2023, pp. 000271-000276, doi: 10.1109/CINTI59972.2023.10382009

[S9] A. Szabó and Á. Pintér, "**Galaxy detection and classification in sky images with neural network,**" 2023 IEEE 23rd International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2023, pp. 000277-000280, doi: 10.1109/CINTI59972.2023.10382059

Irodalomjegyzék

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, ”*Comparison and Evaluation of Clone Detection Tools*”, Transactions on Software Engineering, 33(9):577-591 (2007).

[2] C K. Roy, J. R. Cordy, R. Koscheke. ”*Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach*”, Science of Computer Programming, 44(7):470-495 (2009).

[3] B. Kumar, S. Singh, ”*Code Clone Detection and Analysis Using Software Metrics and NEural Network-A Literature Review*”, Internal Journal of Computer Science Trends and Technology, 3(4):127-132 (2015).

[4] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, ”*Deep learning Code Fragments for Code Clone Detection*”, Automated Software Engineering, ISBN: 978-1-4503-3845-5/16/09, Singapure (2016).

[5] E. Kodhai, S. Kanmani, ”*Method-level code clone detection through LWH (Light Weight Hybrid) approach*”, Journal of Software Engineering Research and Development (2014).

[6] P. Prem, ”*A Review on Code Clone Analysis and Code Clone Detection*”, International Journal of Engineering and Innovative Technology, 2(12):43-46 (2013).

[7] Sun, Y., Ma, L., & Wang, S.”*A comparative evaluation of string similarity metrics for ontology alignment.*” Journal of Information &Computational Science, 12(3), 957-964. (2015). doi: 10.12733/jics20105420

[8] M. Agrawal and D. K. Sharma, ”*A state of art on source code plagiarism detection,*” 2nd International Conference on Next Generation Computing Technologies (NGCT), Dehradun, India, 2016, pp. 236-241, (2016). doi: 10.1109/NGCT.2016.7877421.

[9] S. Zhang, Y. Hu and G. Bian, ”*Research on string similarity algorithm based on Levenshtein Distance,*” 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chongqing, China, 2017, pp. 2247-2251, doi: 10.1109/IAEAC.2017.8054419.

[10] В. И. Левенштейн (1965). ”*Двоичные коды с исправлением выпадений, вставок и замещений символов [Binary codes capable of correcting deletions, insertions, and reversals]*”. Доклады Академии Наук СССР (in Russian). 163 (4): 845–848. Appeared in English as: Levenshtein, Vladimir I. (February 1966). “*Binary codes capable of correcting deletions, insertions, and reversals*”. Soviet Physics Doklady. 10 (8):707–710.

[11] Winkler, W. E.”*String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*”. Proceedings of the Section on Survey Research Methods. American Statistical Association: 354–359. (1990).

[12] N. Cheng, Z. Yu and K. Wang, ”*String similarity computing based on position and cosine,*” *2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC), Macau, China, 2017, pp. 256-261, d*oi: 10.1109/ICEIEC.2017.8076557.

[13] Jaccard, P.”Étude comparative de la distribution florale dans une portion des Alpes et des Jura.” *Bull Soc Vaudoise Sci Nat*, *37*, 547-579. 1901.

[14] Carass, A.; Roy, S.; Gherman, A.; Reinhold, J.C.; Jesson, A.; et al. "Evaluating White Matter Lesion Segmentations with Refined Sørensen-Dice Analysis". *Scientific Reports. 10 (1): 8242. Bibcode:2020NatSR..10.8242C*. 2020, doi:10.1038/s41598-020-64803-w

[15] L. Bergroth, H. Hakonen and T. Raita, ”*A survey of longest common subsequence algorithms,”* Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, A Curuna, Spain, 2000, pp. 39-48, doi: 10.1109/SPIRE.2000.878178.

[16] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. ”*String similarity joins: an experimental evaluation*”. Proc. VLDB Endow. 7, 8 (April 2014), 625–636. doi: 10.14778/2732296.2732299

[17] Chen, H. (2012). ”*String metrics and word similarity applied to information retrieval”.* (Master's thesis, Itä-Suomen yliopisto).

[18] Babenko, M.A., Starikovskaya, T. ”*Computing the longest common substring with one mismatch.*” Probl. Inf. Transm. 47(1), 28–33 (2011). doi: 10.1134/S0032946011010030

[19] HackerRank, Insertion Sort - Part 1. Url:

https://www.hackerrank.com/challenges/insertionsort1/problem (utoljára megtekintve: 2025.06.01)

[20] HackerRank, Insertion Sort - Part 2. Url: https://www.hackerrank.com/challenges/insertionsort2/problem (utoljára megtekintve: 2025.06.01)

[21] HackerRank, Pairs - Url: https://www.hackerrank.com/challenges/pairs/problem (utoljára megtekintve: 2025.06.01)

[22] HackerRank, Simple Array Sum – Url: https://www.hackerrank.com/challenges/simple-array-sum/problem (utoljára megtekintve: 2025.06.01)

[23] M. Crochemore, C. S. Iliopoulos, A. Langiu, and F. Mignosi, "*The longest common substring problem,*" Mathematical Structures in Computer Science, vol. 27, no. 2, pp. 277–295, 2017.

[24] M. Arnold and E. Ohlebusch, "*Linear time algorithms for generalizations of the longest common substring problem,*" Algorithmica, vol. 60, no. 4, pp. 806–818, 2011.

[25] M. Crochemore, A. Langiu, F. Mignosi, and M. Mirisola, "*Longest common substrings, related problems and applications,*" in Biologicla Knowledge, Discovery Handbook. John Wiley & Sons, Inc, 2013, pp. 3–27.

[26] J. R. Thurman, "*Identifying source code clone candidates: A dynamic programming approach based on Levenshtein edit distance and longest common substring algorithms.*" Western Illinois University, 2012.

[27] M. Hajiaghayi, S. Seddighin, and X. Sun, "*Massively parallel approxima-tion algorithms for edit distance and longest common subsequence,*" in Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2019, pp. 1654–1672.

[28] T. T. Ngoc, C. M. T. Le Van Dai, and C. M. Thuyen, "*Support vector regression based on grid search method of hyperparameters for load forecasting,*" Acta Polytechnica Hungarica, vol. 18, no. 2, pp. 143–158, 2021

[29] B. Tusor and A. R. Várkonyi-Kóczy, "*Fast regular and interval-based classification, using parsits,*" Acta Polytechnica Hungarica, vol. 18, no. 6, pp. 107–125, 2021.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, „*Introduction to algorithms*". MIT press, 2022.

[31] A. Amir, P. Charalampopoulos, S. P. Pissis, and J. Radoszewski, "*Dy-namic and internal longest common substring,*" Algorithmica, vol. 82, no. 12, pp. 3707–3743, 2020.

[32] Thomas Schmidt and Jens Stoye. "*Quadratic time algorithms for finding common intervals in two and more sequences.*" Proc of CPM 2004 LNCS, 3109:347–358, 2004. doi:10.1007/978-3-540-27801-6_26.

[33] Beata Bylina and Jaroslaw Bylina. "*Strategies of parallelizing nested loops on the multicore architectures on the example of the wz factorization for the dense matrices.*" Proceedings of the Federated Conference on Computer Science and Information Systems, pages 629–639, 2015. doi: 10.15439/2015f354.

[34] Beata Bylina and Jaroslaw Bylina. "*Parallelizing nested loops on the intel xeon phion the example of the dense wz factorization.*" Proceedings of the Federated Conference on Computer Science and Information Systems, 8:655–664, 2016. doi:10.15439/2016f436.

[35] Adrian Jackson and Orestis Agathokleous. "*Dynamic loop parallelisation.*" arXiv:1205.2367, 2012.

[36] M.E. Wolf and M.S. Lam. "*A loop transformation theory and an algorithm to maximize parallelism.*" IEEE Transactions on Parallel and Distributed Systems, 2(4):452–471, 1991. doi: 10.1109/71.97902.

[37] T. H. Tzen and L. M. Ni, "*Dependence uniformization: a loop parallelization technique,*" in IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 5, pp. 547-558, May 1993, doi: 10.1109/71.224217

[38] H. Blinn, "*How to solve a quadratic equation?,*" in IEEE Computer Graphics and Applications, vol. 25, no. 6, pp. 76-79, Nov.-Dec. 2005, doi: 10.1109/MCG.2005.134.

[39] Renáta, N. Az automatizált szövegfeldolgozás szociológiai lehetőségei. 2004.

[40] Gosal A. and Septifani R. (2025). Implementation of Machine Learning in the Classification of Text Created by Humans or Artificial Intelligence. In Proceedings of the 1st International Conference on Research and Innovations in Information and Engineering Technology - Volume 1:

RITECH; ISBN 978-989-758-784-9, SciTePress, pages 45-52. doi: 10.5220/0014269100004928

[41] Lu, Jiaxin. "Text vectorization in sentiment analysis: A comparative study of TF-IDF and Word2Vec from Amazon Fine Food Reviews." *ITM Web of Conferences*. Vol. 70. EDP Sciences, 2025.

[42] Anand, Lawal. (2022). Convolutional Neural Networks for Natural Language Processing: A Review. doi: 10.13140/RG.2.2.28888.15367.

[43] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[44] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019.

[45] Liu, Yinhan, et al. "Roberta: A robustly optimized bert pretraining approach." *arXiv preprint arXiv:1907.11692* (2019).

[46] Brown, Tom, et al. "Language models are few-shot learners." *Advances in neural information processing systems* 33 (2020): 1877-1901.

[47] OpenAI, R. "Gpt-4 technical report. arxiv 2303.08774." *View in Article* 2.5 (2023): 1.

[48] Ji, Ziwei, et al. "Survey of hallucination in natural language generation." *ACM computing surveys* 55.12 (2023): 1-38.

[49] Sima D., Schmuck B., Szollősi S., Miklós Á. ”*Intelligent short text assessment in eMax,*” Proceedings of the Eight Africon Conference, Windhoek, South Africa, 26-28 September 2007, Paper 4401593.

[50] Frtunić Gligorijević, M., Stoimenov, L., Vojinović, O., Milentijević, I. ”*Towards flexible short answer questions in the Moodle Quiz*.” In: Konjović, Z., Zdravković, M., Trajanović, M. (Eds.) ICIST 2016 Proceedings, pp.5-9, 2016

[51] Sukkarieh, J., Pulman, S., & Raikes, N. *”Automatic marking of short, free text responses.”* 2005.

[52] Marzouq, A. A. (2024). „*Utilizing Cognitive Flexibility Theory to*

Promote EFL Postgraduates" Academic Reading Comprehension and Selective Attention. BSU-Journal of Pedagogy and Curriculum, 3(6), 133-190.

[53] Chui, G. Discourse "*Processes Shallow Processing and Underspecification*". https://doi.org/10.1207/S15326950DP4202_1

[54] Mayfield, E., Adamson, D., Woods, B., Miel, S., Butler, S., & Crivelli, J. "*Beyond automated essay scoring: Forecasting and improving outcomes in middle and high school writing.*" In Companion proceedings of the 8th international conference on learning analytics and knowledge (pp. 1-8). 2018.

[55] Ye, C., Yang, Y., Fermuller, C., & Aloimonos, Y. (2017). "*On the importance of consistency in training deep neural networks*". arXiv preprint arXiv:1708.00631. 2017.

[56] Dudek-Dyduch, E., Tadeusiewicz, R., & Horzyk, A. "*Neural network adaptation process effectiveness dependent of constant training data availability*." Neurocomputing, 72(13-15), 3138-3149. 2009.

[57] Document-specific word2vec, Training corpus, url: https://kbpedia.org/use-cases/document-specific-word2vec-training-corpuses/, (utoljára megtekintve: 2025.06.01).

[58] Norvig P., "*How to Write a Spelling Corrector,*", 2016. url: https://norvig.com/spell-correct.html (utoljára megtekintve: 2025.06.01)

[59] Liu J., Meng F., Yong Z., Liu B. "*Character-Level neural networks for short text classification*," 2017 International Smart Cities Conference (ISC2), Wuxi, China, 14-17 Sept 2017.

[60] K. N. M. Ngafidin, S. T. Safitri and D. M. Kusumawardani, "*A Study on SME Traditional Food Product Feedback by Leveraging Word2vec,*" 2024 Beyond Technology Summit on Informatics International Conference (BTS-I2C), Jember, East Java, Indonesia, 2024, pp. 71-76, doi: 10.1109/BTS-I2C63534.2024.10942267.

[61] Huy N., Minh-Le N. "*A Deep Neural Architecture for Sentence-level Sentiment Classification in Twitter Social Networking*", Conference of the Pacific Association for Computational Linguistics, Yangon, Maynmar, August 16 – 18, 2017

[62] Huy T. N., Minh-Le N. "*Sentence Modeling with Deep Neural*

Architecture using Lexicon and Character Attention Mechanism for Sentiment Classification", Proceedings of the The 8th International Joint Conference on Natural Language Processing, Taipei, Taiwan, November 27 - December 1, 2017, pp. 536-544.

[63] Kim Y. "*Convolutional neural networks for sentence classification,*" Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, Doha, Qatar, 25-29 October 2014, pp. 1746–1751

[64] Severyn A., Moschitti A. "*Learning to rank short text pairs with convolutional deep neural networks,*" Proceedings of the 38th Internal ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, 9-13 Aug 2015, pp. 373–382

[65] T. Almeida and J. Hidalgo. "SMS Spam Collection," UCI Machine Learning Repository, 2011. [Online]. Available: https://doi.org/10.24432/C5CC84. (utoljára megtekintve: 2026.04.01)

[66] D. Kotzias. "Sentiment Labelled Sentences," UCI Machine Learning Repository, 2015. [Online]. Available: https://doi.org/10.24432/C57604. (utoljára megtekintve: 2026.04.01)

[67] N. D. Q. Bui, L. Jiang, Y. Yu, "*Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks*", arXiv:1710.06159, 2017.

[68] S. Gilda, "*Source code classification using Neural Networks*", IEEE 14th International Joint Conference on Computer Science and Software Engineering, 12-14 July 2017.

[69] H. T. Nguyen, M. L. Nguyen, "*Sentence Modeling with Deep Neural Architecture using Lexison and Character Attention Mechanism for Sentiment Classification*", Proceedings of the 8th International Joint Conference on Natural Language Processing, Taipei, Taiwan, pp. 536-544, November 27 – December 1, 2017

[70] Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." *Findings of the association for computational linguistics: EMNLP 2020*. 2020.

[71] Guo, Daya, et al. "Graphcodebert: Pre-training code representations with data flow." *arXiv preprint arXiv:2009.08366* (2020).

[72] Lozhkov, Anton, et al. "Starcoder 2 and the stack v2: The next

generation." *arXiv preprint arXiv:2402.19173* (2024).

[73] Roziere, Baptiste, et al. "Code llama: Open foundation models for code." *arXiv preprint arXiv:2308.12950* (2023).

[74] Martinez-Gil, Jorge. "Improving source code similarity detection through graphcodebert and integration of additional features." *arXiv preprint arXiv:2408.08903* (2024).

[75] Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." *Iclr* 1.2 (2022): 3.

[76] M. White, M. Tufano, C. Vendome and D. Poshyvanyk, ”*Deep learning code fragments for code clone detection,*” 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 2016, pp. 87-98.

[77] L. Jiang, G. Misherghi, Z. Su and S. Glondu, ”*DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones,*” 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 96-105, doi: 10.1109/ICSE.2007.30.

[78] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2018). ”*code2vec: Learning Distributed Representations of Code.*” https://arxiv.org/abs/1803.09473

[79] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, Z. Jin, “*Building Program Vector Representations for Deep Learning*”, Springer Knowledge Science, Engineering and Management, pp. 547-553, 2015.

[80] HackerRank, A Very Big Sum – Url: https://www.hackerrank.com/challenges/a-very-big-sum/problem (utoljára megtekintve: 2025.06.01)

[81] HackerRank, Big Sorting – Url: https://www.hackerrank.com/challenges/big-sorting/problem (utoljára megtekintve: 2025.06.01)

[82] Z. Djuric, D. Gasevic, “*A Soure Code Similarity System for Plagiarism Detection*”, in The Computer Journal. vol 56. pp. 70-78, January 2013.

[83] Tokenization, url: https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html (utoljára megtekintve: 2025.06.01)

[84] T. Mikolov, K Chen, G. Corrado, J. Dean, "*Efficient Estimation of Word Representations in Vector Space*", Proceedings of the International Conference on Learning Representations, pp 1-12, 2013.

[85] M. E. Khademi, M. Fakhredanesh, S. M. Hoseini, "*Conceptual Text Summerizer: A New Model In Continuous Vector Space*", arXiv:1710.10994, 2017.

[86] word2vec project website, url: https://code.google.com/archive/p/word2vec/ (utoljára megtekintve: 2025.06.01)

[87] K. Yoon, "*Coonvolutional Neural Networks for Sentence Classification*", Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, pp. 1746-1751, 25 August 2014.

[88] D. Britz, "Understanding Convolutional Neural Networks for NLP", url: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/ (utoljára megtekintve: 2025.06.01)

[89] L. Maaten, G. Hinton, *"Visualizing Data using t-SNE"*, Journal of Machine Learning Research 9, pp 2579-2605 (2008).

[90] C.-Y. Lee, P. W. Gallagher, Z. Tu, "*Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree*", arXiv:1509.08985, 2015.

Ábrajegyzék

Táblázatjegyzék

Algoritmusjegyzék